ISSN 0265-2919

90p 63

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO

An ORBIS Publication

IR£1.15 Aus $2.15 NZ $2.65 SA R2.45 Sing $4.50

CONTENTS

APPLICATION

SEARCH PARTY Developing techniques for intelligently 'searching' a maze is an important area of AI research. We take a look at a few methods and provide an example program — 1241

HARDWARE

COMPETITIVE COMPATIBLE The Tandy 1000 is a true IBM PC-compatible which offers all the advantages of the market leader at a considerably lower price — 1249

SOFTWARE

ELEMENTAL FORCES Archon is a unique hybrid game that requires the strategic skills of a chess player and the fast finger responses of an arcade enthusiast — 1260

COMPUTER SCIENCE

GOTO END We conclude our PASCAL tutorial series with a look at a few further aspects that give the language its powerful reputation — 1246

JARGON

FROM PARAMETER TO PASS A weekly glossary of computing terms — 1252

PROGRAMMING PROJECTS

THE NEW WORLD We must now write a module for the simulation game to deal with our first encounter with the inhabitants of the New World — 1244

MACHINE CODE

FRAME WORK The first part of a project to develop a 3-D graphics package for the Commodore 64. This week, we consider some of the floating point mathematics used in the program — 1256

WORKSHOP

COPY WRITER We begin a new project: the construction of a digital tracer to be interfaced with the BBC Micro. Here, we provide the plan and a parts list — 1253

REFERENCE CARD Another part of the Commodore 64's memory map — INSIDE BACK COVER

Next Week

- **We begin a new series on the most popular operating systems used on micros, starting with a beginner's quide to CP/M**
- **Our series on AI focuses on how the newest machines have been taught to play one of the oldest games in the world. We look at the techniques behind programming computers to be Chess Grandmasters**
- **To coincide with our artificial intelligence series, we begin a new tutorial course on PROLOG — a language frequently used in AI research**

QUIZ

1) When searching for the centre of a maze, what problem are we likely to encounter if we use a 'hill climbing' strategy?
2) What function is performed by the potentiometers on the digital tracer?
3) Is there any difference between the BIOS on the Tandy 1000 and that of the IBM PC?
4) What parameters are used by Commodore BASIC to retrieve a string?

Answers To Last Week's Quiz

1) Rosenblatt's Perceptron was an early device that could recognise simple patterns.
2) A query language is a system where commands are entered in response to simple prompts.
3) Although the Z80B can address only 64 Kbytes of memory directly, by using bank switching the Wren Executive is capable of accessing up to 400 Kbytes.
4) Position eight in a paper tape system of input is used for 'parity checking' to ensure the data has been received correctly.

Editor Stephen Cooke; **Art Editor** Claudia Zeff **Deputy Editor** Steve Colwill; **Production Editor** Bobby Pickering; **Designer** Julian Dorr; **Art Assistant** Liz Dixon; **Staff Writer** Steve Malone; **Sub Editor** Jon Kaye; **Contributors** Richard Forsythe, Nick Walsh, Dr Antonia Jones, Terry Barrett, Steve Malone, Anthony Ginn, Steve Colwill; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Alastair Gourlay; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1985; © Orbis Publishing Ltd 1985: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derby**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders. **UK/EIRE** – Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Price: SA R2.45. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Price: AusS2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Price: NZ$2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited. Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY MARCUS WILSON-SMITH

The Rat Race
The diagram shows three possible routes through the maze using three different search strategies: random, systematic and heuristic. In this example, the heuristic method finds the goal quicker than the other two methods, although it would be possible to design a maze in which this was not necessarily the case. In practice, mazes are often best solved by combining heuristic and exhaustive (one in which all points are considered) search methods

STEVE CROSS

SEARCH PARTY

Exploring the concept of 'searching' (i.e. for a solution to a problem) has been an important part of research into artificial intelligence. We discuss some fundamental ideas about search techniques, and provide a program that implements three advanced methods of search to solve a traditional 'rat in a maze' problem.

Imagine three rats being let loose in a maze, somewhere inside of which is a bowl full of appetising rat pellets. One of the rats blunders around for a few minutes and then falls asleep (the callous experimenter has been spiking the animal's drinking water with gin). The second is more methodical: it puts its left forepaw against a wall and continues to take left turns at every fork, backing up when it hits a dead end. Eventually it reaches its goal, but by then the third rat has eaten all the food.

The third rat has a keen sense of smell. At several points in its search, it sniffs the air and takes the path it feels will lead closer to the delicious aroma. It is possible to devise a maze that would fool this rat (just as it is possible to devise one to confound the wall follower) but most of us would agree that this kind of search strategy is the most intelligent of the three.

Search is a key concept in the realm of artificial intelligence. Whether you are 200 feet under the Caribbean in a scuba outfit looking for sunken treasure or sitting in a train scratching your head over '4 Across', you are searching for something. The metaphor equating problem solving with search has proved very fruitful in AI because many different problems can be dealt with using search techniques. Our three rats illustrate different kinds of search strategy:

- Random Search (the 'Drunkard's Walk')
- Exhaustive Search (Systematic Enumeration)
- Heuristic Search (Guided Exploration)

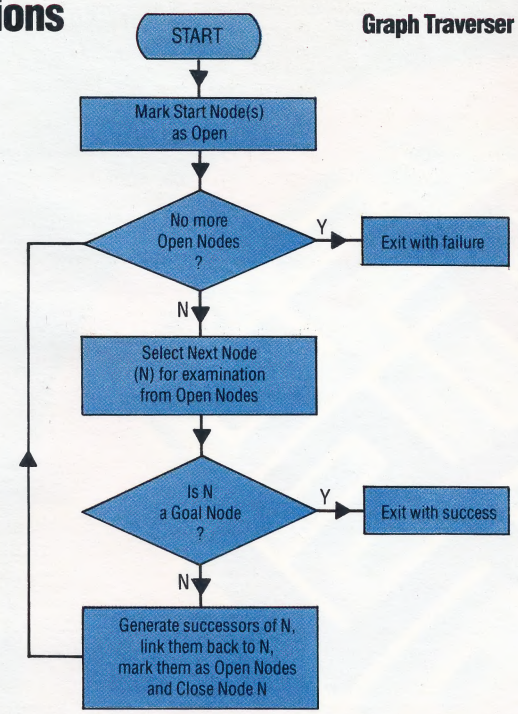
The third is considered a more intelligent approach to the problem than the others because it will generally require less effort to reach the solution. But all heuristic methods rely on some means of knowing when the search is approaching its goal (the rat's sense of smell, for instance). Without knowledge, you cannot be intelligent: you have to rely on systematic enumeration.

Finding a route through a maze is a typical

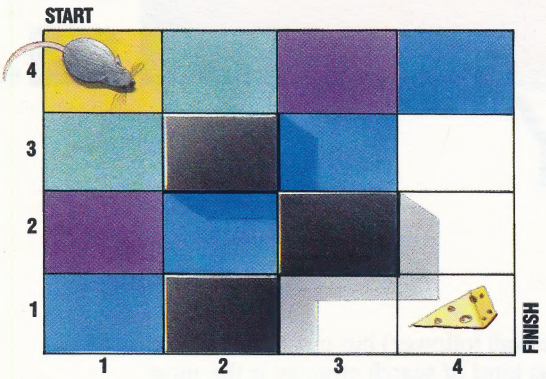
Searching Questions

Most AI search methods involve generating trees from the layout of the maze to be traversed. The tree shown here is generated from the maze by taking the starting point as the root node, and growing further generations by considering all the possible moves from the node in question. Note that diagonal moves are not allowed. The flowchart outlines the basic algorithm for generating the tree structure

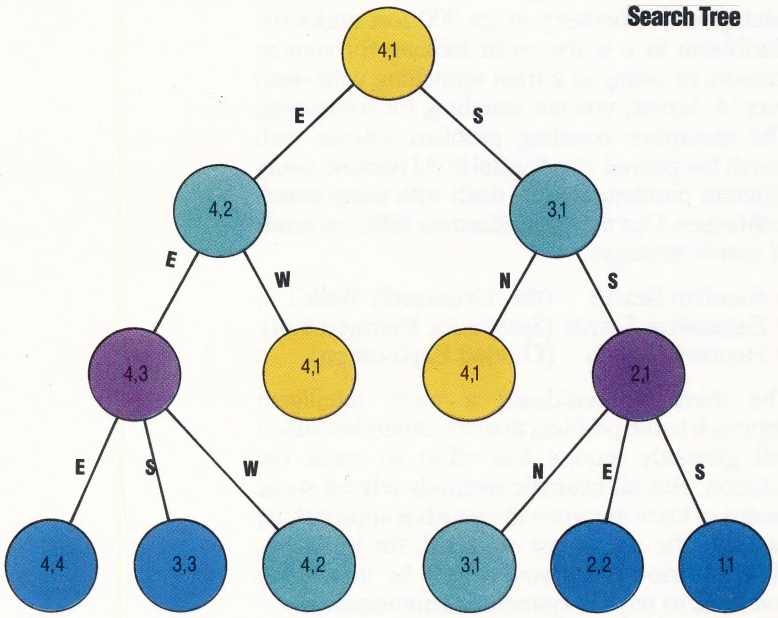
Graph Traverser


START
Mark Start Node(s) as Open
No more Open Nodes ?
Y
Exit with failure
N
Select Next Node (N) for examination from Open Nodes
Is N a Goal Node ?
Y
Exit with success
N
Generate successors of N, link them back to N, mark them as Open Nodes and Close Node N

A Simple Maze



Search Tree



KEVIN JONES

search problem. (It is important to realise, however, that many problems which are non-spatial, such as integrating a symbolic mathematical expression, can be treated using the same search framework.) There are two ways by which the quality of a search method may be judged:

- How long does it take to find a route?
- Is this the most economic route?

Ideally, we want a method that finds the optimal solution in the least possible time. In practice, however, we may be forced to compromise by either accepting a suboptimal solution in a hurry or by taking a long time to find the very best solution.

Most artificial intelligence search strategies are built according to a common plan, which allows more or less weight to be placed on one or the other of these performance criteria. The general graph traversal algorithm diagram shown outlines a family of search methods from which a particular member can be selected by choosing how to fill the central box — or, how to decide which node to examine next. The terms 'open' and 'closed' refer to the status of any node of the path from the beginning to the destination. Open nodes are candidates for examination; closed nodes have already been examined. The open nodes are on a kind of waiting list, and the key to efficiency is the order in which this list is processed.

A SEARCH TREE

The search proceeds by growing a tree. The simple four by four maze as shown depicts the robot rat in the top left square and the food in the bottom right. At each square, the rat has a choice of up to four moves (north, south, east or west), though some may be blocked. If we draw the options open to the rat at the beginning, you will see the tree-like structure of the search process.

From the initial square at (4,1) the rat can only move east to (4,2) or south to (3,1). From (4,2) it can move east or west and from (3,1) it can move either north or south. If the rat moves east and then west (or south and then north) it will arrive back at the starting point, which is not terribly clever, but it does show that the same square can appear at different nodes in the tree, corresponding to the various ways of arriving at them.

One systematic method of exploring the tree is termed 'breadth-first search'. It investigates nodes in order of proximity to the start, or root. Thus it considers every sequence of N moves (at level N in the tree) before any sequence of N+1 moves. It is bound to find a route that takes the fewest steps, and as long as each step is equally costly, will prove to be the minimum-cost solution. But it may take a long time to find it. As the tree becomes wider and deeper, the time required to find the goal increases exponentially.

To improve on a breadth-first search requires a source of heuristic information about how far away the goal is. We could use the Manhattan

Distance as such a measure. On Manhattan Island, in New York State, most streets intersect at right angles. To get from A to B you must move so many blocks north or south and so many east or west. Similarly, in the maze, an intelligent rat can calculate how many squares away the target is.

By knowing when a search is getting near a solution, it can be speeded up. The rat in our opening paragraphs followed a simple rule: always move closer to the goal. This is the 'hill-climbing' strategy. It is so-called because it can be compared with finding the summit of a mountain in the fog by always moving upward. It can be much faster than a breadth-first search but it is not guaranteed to find the optimal route. It may get stuck on a local peak (because the search is looking for *any* summit).

A better method is a compromise between breadth-first and simple hill-climbing. This is called the A-Star (A*) algorithm and it selects the node to examine next on the basis of: HD + SF, where HD is the heuristic estimate of distance remaining and SF is the distance covered so far. The better the estimate of HD, the more effective the search. If HD errs, it should underestimate rather than overestimate.

In our example program, we use one program framework to implement all three methods, with only minor modifications. The only differences are made evident in the choice of which node to examine next:

Breadth-First Search — pick lowest SF
Hill-Climbing — pick lowest HD
A* Algorithm — pick lowest HD+SF

Maze Search Program

```
 1000 REM ******************************
 1010 REM ** Listing 2.1 :             **
 1020 REM ** MAZE SEARCH PROGRAM       **
 1030 REM ******************************
 1040 MODE7
 1050 MH=17: MW=25: REM Maze Height & Width
 1060 SI=256 : REM tree limits.
 1070 WA=1: RR=2: FO=3: DN=4: BL=5
 1080 W1=1 : REM weight for SF
 1090 W2=2 : REM weight for HD
 1100 DIM M(MH+1,MW+1) : REM the maze
 1110 DIM C$(5) : REM maze characters
 1120 DIM P(SI),S(SI),N(SI),H(SI)
 1130 REM Path,Steps,Node,Heuristic-distance.
 1140
 1150 REM -- Rat in the Maze:
 1160 GOSUB 1360 : REM make the maze
 1170 NC=0 : REM no. of nodes examined
 1180 K=0 : REM counter
 1190 GOSUB 1660 : REM clear all paths
 1200 N(1)=2*MW+2 : REM 1st open node
 1210 S(1)=0: H(1)=FR-1 + (FC-1)
 1220 P(1)=0: REM no predecessor
 1230 REM -- Main loop:
 1240 REM **** MAIN LOOP ****
 1250 GOSUB 1770 : REM pick next node S
 1260 NC=NC+1
 1270 PRINT TAB(0,22);NC,SR,SC,H(S);"          "
 1280 GOSUB 1880 : REM generate successors
 1290 IF (FR<>SR OR FC<>SC) AND NC<300 THEN 1240
 1300 PRINT TAB(0,22);"Search finished!"
 1310 IF FR=SR AND FC=SC THEN GOSUB 2290 : REM ret
race steps
 1320 IF NC>300 THEN PRINT "Failed!"
 1330 PRINT NC;" nodes examined."
 1340 END
 1350 :
 1360 REM -- Routine to create maze:
 1370 FOR P=1 TO MH+1
 1380 FOR R=1 TO MW+1
 1390 IF RND(1) < 0.28 THEN M(P,R)=WA ELSE M(P,R)=
BL
 1400 IF P=3 OR R=3 THEN M(P,R)=BL
 1410 IF P=1 OR R=1 THEN M(P,R)=WA
 1420 IF P>MH OR R>MW THEN M(P,R)=WA
 1430 NEXT: NEXT
 1440 FR=2+INT(RND(1)*(MH-1)): REM food row
 1450 FC=4+INT(RND(1)*MW-3): REM food col
 1460 M(FR,FC)=FO
 1470 M(2,2)=RR : REM robot rat
 1480 C$(BL)=" "
 1490 C$(WA)=CHR$255 : REM blob
 1500 C$(DN)=" "
 1510 C$(RR)="R"
 1520 C$(FO)="f"
 1530 GOSUB 1560 : REM display it
 1540 RETURN
 1550 :
 1560 REM -- Maze display routine:
 1570 CLS
 1580 FOR R=1 TO MH+1
 1590 FOR C=1 TO MW+1
 1600 PRINT TAB(C,R); C$(M(R,C));
 1610 NEXT
 1620 PRINT
 1630 NEXT
 1640 RETURN
 1650 :
 1660 REM -- Tree-clearing routine:
 1670 DD=9999 : REM dead
 1680 FOR Q=1 TO SI
 1690 P(Q)=0
 1700 S(Q)=DD
 1710 N(Q)=0
 1720 H(Q)=DD
 1730 NEXT
 1740 NN=2 : REM next free node
 1750 RETURN
 1760 :
 1770 REM -- Pick best node S:
 1780 S=1: BN=DD
 1790 FOR I=1 TO SI
 1800 V=S(I)*W1 + ABS(H(I))*W2
 1810 IF V<BN AND H(I)>=0 THEN S=I: BN=V
 1820 NEXT
 1830 IF S=1 THEN PRINT TAB(0,20);"Exploring...."
 1840 SR=INT(N(S)/MW)
 1850 SC=N(S)-MW*SR
 1860 RETURN
 1870 :
 1880 REM -- Routine to generate successors:
 1890 IF H(S)=0 THEN RETURN : REM done.
 1900 REM -- North:
 1910 Y=SR-1 : X=SC
 1920 IF Y>1 THEN GOSUB 2090
 1930 REM -- East:
 1940 Y=SR : X=SC+1
 1950 IF X<=MW THEN GOSUB 2090
 1960 REM -- South:
 1970 Y=SR+1 : X=SC
 1980 IF Y<=MH THEN GOSUB 2090
 1990 REM -- West:
 2000 Y=SR : X=SC-1
 2010 IF X>1 THEN GOSUB 2090
 2020 REM -- also close node S:
 2030 H(S) = -H(S)
 2040 IF H(S)>0 THEN PRINT "Ugh!"
 2050 PRINT TAB(SC,SR);" ";
 2060 M(SR,SC)=DN
 2070 REM blanks cell on screen.
 2080 :
 2090 REM -- Routine to open 1 node:
 2100 IF M(Y,X)=DN THEN RETURN
 2110 IF M(Y,X)=WA THEN RETURN
 2120 REM -- 1st find free location:
 2130 NX=0
 2140 REM ** FIND LOCATION LOOP
 2150 IF S(NN)<>DD THEN NX=NX+1: NN=NN+1
 2160 IF NN>SI THEN NN=1
 2170 IF NX>SI THEN PRINT "Full up!": STOP
 2180 IFS(NN)<>DD THEN 2140
 2190 REM -- Now open it:
 2200 XY=X + Y*MW
 2210 N(NN)=XY
 2220 P(NN)=S
 2230 S(NN)=S(S)+1
 2240 H(NN)=ABS(Y-FR) + ABS(X-FC)
 2250 PRINT TAB(X,Y);"+";
 2260 REM shows it on screen.
 2270 RETURN
 2280 :
 2290 REM -- Path-retracing routine:
 2300 ST=S(S)
 2310 FOR Q=1 TO 10000:NEXT Q:GOSUB1560
 2320 PRINT TAB(FC,FR);C$(FO);
 2330 REM ** PRINT PATH **
 2340 S=P(S) : REM parent node
 2350 XY=N(S) : REM coords.
 2360 Y=INT(XY/MW)
 2370 X=XY-Y*MW
 2380 M(Y,X)=RR : REM rat's footprint!
 2390 PRINT TAB(X,Y);"*";
 2400 IF S>0 THEN 2330
 2410 PRINT TAB(2,2);C$(RR)
 2420 PRINT TAB(0,22);"Path of ";ST;" steps."
 2430 PRINT NC;" nodes closed."
 2440 RETURN
```

Maze Lines

The maze is contained in a two-dimensional array M(,), and the data structures for the search tree are held in arrays P(), S(), N() and H(). The program combines cost-so-far and estimated-cost-to-goal factors which can be weighted by altering the values W1 and W2 in lines 1080 and 1090. With the values as listed, the program will follow a hill-climbing strategy but you may wish to experiment. The heuristic method used in this program is based on Manhattan Distance which is very accurate in this example. Thus, tipping the balance towards the heuristic method by making W2 > W1 speeds up the search

Basic Flavours

The program is written for the BBC Micro. For the Commodore 64 and Spectrum make the following changes:

Commodore 64:

Replace all occurrences of PRINT TAB(X,Y);"MESSAGE" with P=X:Q=Y:GOSUB

```
3000 :PRINT"MESSAGE"
3000 REM ** TAB ROUTINE **
3010 PRINT CHR$(19);
3020 PRINT|LEFT$(DW$,Q);
     TAB(P);
3030 RETURN

1040 FOR I=1 TO 25:
     DW$=DW$+CHR$(17):
     NEXT I
1490 C$(WA)=CHR$(102)
1570 PRINT CHR$(147)
```

Spectrum:

Replace all occurrences of TAB(X,Y); with AT X,Y;

```
1490 LET C$(WA)=CHR$(143)
```

Delete line 1040

	(1)	(2)	(3)	
T$()	Pearls	Carvings	Spices	Descriptions of goods offered by Chief.
V1()	2gp	2gp	1gp	Market prices of goods when ship left port.
V2()	?	?	?	Market prices of goods when ship returns.
EQ(,) 1 Bag of Salt	0.5	0.5	1	Exchange rates of goods to be traded.
1 Bale of Cloth	5	5	10	
1 Jewel	3	3	6	
1 Knife	2	2	4	

THE NEW WORLD

Your ship has now reached the New World without, one hopes, having encountered too many difficulties. If you can avoid starting a war with the local inhabitants, it will then be possible to engage in some lucrative trading, allowing you to return home to port.

As we reach the final stage of the game, the voyage itself will end as you attempt to land successfully in the New World and trade the goods you purchased at the outset of the game. These will hopefully bring you a profit when you return to port and sell your newly acquired commodities. But the danger is not over yet — you have never met the inhabitants of this new land, and they can be either friendly or hostile. Therefore, care must be taken to avoid any actions that might cause antagonism between them and your crew.

When the main voyage loop is terminated, line 891 calls subroutine 10000, which deals with your arrival in the New World. As the ship approaches the shore, several parties of armed local inhabitants paddle out to investigate. You must now decide whether to risk having the ship attacked by refusing to shoot, or else opening fire and possibly causing a war.

Line 10015 checks the second element of the supplies array, OA(2), which indicates the number of guns on board. If there are guns available, then you must decide on which course of action to take. Line 10022 waits for a reply; if you type in a 'yes' and use your guns, many of the natives will be killed. However, such a course of action is likely to provoke the survivors to return to the ship during the night and set it ablaze — thus ending the game. Obviously, unprovoked aggression does not pay. (Line 10044 has been inserted to prevent you from continuing the game after this unwise decision.)

If you decide not to shoot, there will be, of course, no need for the natives' retribution, and line 10026 will send the program to line 10050. In this section, the natives board the ship peacefully and greet the crew, who are then taken to the village where they meet the chief. The chief turns out to be quite friendly and has bartered with representatives of the Old World on previous occasions. After being fed and rested, the trading can begin on the following day (dealt with in a separate module). We will discuss this module in the next instalment. But, before trade can begin, we must make some initial arrangements.

THE TRADING ARRAYS

The prices of pearls, spices and carvings were determined when the ship left port, but they have changed since then.

To facilitate trading, several new arrays are created. Line 60 DIMensions the array T$(), which contains descriptions of the three types of goods offered by the chief: pearls, carvings and spices. Line 61 DIMensions an array, V1(), containing the market values of these goods when the ship left port.

Pearls and carvings were selling for two gold pieces each and spices could be bought at one gold piece per gram. Line 62 DIMensions the array V2() containing the prices of the goods when the ship returns to port. But because the market values of the goods fluctuate, a random element influences the setting of the final price. The price of V2(1), representing pearls, will be set at two or $2\frac{1}{2}$ gold pieces by the expression at line 62. Similar expressions on the same line set the price of carvings at one, two or three gold pieces each; spices will be either two or $2\frac{1}{2}$ pieces per gram.

When trading takes place, goods are not bought and sold for gold, but are swapped at agreed exchange rates — one batch of goods for another. The chief offers the ship's captain an amount of pearls, carvings and spices for each lot of goods the captain offers. For simplicity's sake, we will assume that the New World traders have unlimited supplies of the goods they will be trading. The quantities of these are determined by the quantities of goods on the ship.

A two-dimensional array is DIMensioned and values are assigned to it in lines 64 to 68, with the rates of exchange for each commodity. The first subscript corresponds to the four items that you will be trading: salt, cloth, knives and jewels. The second subscript represents the three items the natives will be trading with: pearls, carvings and spices, with the rates of exchange being specified in three lines of the program.

Line 64 sets up the exchange rates for bags of salt, the first element in the first subscript. Pearls are the first element in the second subscript and EQ(1,1) is assigned a value that gives the ratio of pearls to bags of salt. Setting EQ(1,1) to 0.5 sets the exchange rate at half a pearl to one bag of salt. Similarly, EQ(1,2) is assigned a value corresponding to the intersection of the first element in the first subscript and the second element in the second subscript — salt and carvings. When EQ(1,2) is set to 0.5, the exchange rate is one carving per two bags of salt. EQ(1,3)=1 sets the rate of one gram of spice, the third element in the second subscript, for one bag of salt.

The exchange rates for the second element in the first subscript — bales of cloth — are determined in line 66; one bale is worth five pearls, five carvings or 10 grams of spice. Line 67 deals with the third element of the first subscript, knives, which are worth three pearls, three carvings or six grams of spice each. Jewels, the fourth element, are dealt with in line 67, and may be exchanged for two pearls, two carvings or four grams of spice.

An array, AO(3), is DIMensioned at line 69 to store the amounts of pearls, carvings and spices acquired during the trading. This should not be confused with the array OA, which stored the amounts of trading goods purchased at the start of the voyage.

The mechanics of the trading that uses the array values set up in this module will be dealt with in the next instalment.

Module Eleven: Arrival

Dimensioning The Trading Arrays

```
60 DIMT$(3):T$(1)="PEARLS":T$(2)="CARVINGS":T$
(3)="SPICES"
61 DIMV1(3):V1(1)=2:V1(2)=2:V1(3)=1
62 DIMV2(3):V2(1)=2+(INT(RND(1)*1)/2):V2(2)=2+
(INT(RND(1)*3)-1)
63 V2(3)=2+(INT(RND(1)*1)/2)
64 DIMEQ(4,3)
65 EQ(1,1)=0.5:EQ(1,2)=0.5:EQ(1,3)=1
66 EQ(2,1)=5:EQ(2,2)=5:EQ(2,3)=10
67 EQ(3,1)=3:EQ(3,2)=3:EQ(3,3)=6
68 EQ(4,1)=2:EQ(4,2)=2:EQ(4,3)=4
69 DIMAO(3)
```

Addition To Main Program Body

```
890 REM ARRIVAL AT NEWWORLD
891 GOSUB10000
```

Arrival Subroutine

```
10000 REM ARRIVAL AT NEWWORLD
10001 PRINTCHR$(147):GOSUB9200
10005 S$="YOU ARRIVE AT THE NEW WORLD*":GOSUB9100
10006 PRINT:GOSUB9200
10007 S$="AS YOU APPROACH THE SHORE*":GOSUB9100
10009 S$="NATIVES PADDLE OUT TO MEET YOU*":GOSUB91
00
10010 PRINT:GOSUB9200
10015 IFOA(2)=0THEN10050
10017 S$="THEY LOOK FIERCE AND ARE ARMED!!*":GOSUB
9100
10018 PRINT:GOSUB9200
10020 S$="DO YOU OPEN FIRE? (Y/N)*":GOSUB9100
10022 INPUTI$:I$=LEFT$(I$,1)
10024 IFI$<>"N"ANDI$<>"Y"THEN10022
10026 IFI$="N"THEN10050
10028 PRINT:GOSUB9200
10030 S$="MANY NATIVES ARE KILLED*":GOSUB9100
10032 S$="BUT DURING THE NIGHT*":GOSUB9100
10034 S$="OTHERS RETURN*":GOSUB9100
10036 S$="AND BURN YOUR SHIP!!!!*":GOSUB9100
10038 PRINT:PRINT:GOSUB9200
10040 S$="        GAME OVER*":GOSUB9100
10042 END
10044 GOTO10042
10050 S$="THEY TAKE YOU TO MEET THEIR*":GOSUB9100
10052 S$="CHIEF. HE HAS MET YOUR RACE*":GOSUB9100
10054 S$="BEFORE AND IS FRIENDLY.*":GOSUB9100
10056 GOSUB9200
10058 S$="THE CREW ARE FED AND RESTED*":GOSUB9100
10060 PRINT:GOSUB9200
10062 S$="THE NEXT DAY TRADING BEGINS*":GOSUB9100
10064 PRINT:GOSUB9200
10066 S$=K$:GOSUB9100
10068 GETI$:IFI$=""THEN10068
10069 RETURN
```

Basic Flavours

Spectrum:

Replace V1() by B(), V2() by D(), EQ(,) by Q(,) and AO() by E() throughout and make the following changes:

```
60    DIM T$(3,9)
10001 CLS:GO SUB 9200
10022 INPUT I$:LET I$=I$( TO 1)
10068 LET I$=INKEY$:IF I$= "" THEN GO TO 10068
```

BBC Micro:

Make the following changes:

```
10001 CLS:GOSUB 9200
10068 I$=GET$
```

IAN McKINNELL ON THE MACINTOSH

GOTO END

In this concluding instalment of our series on PASCAL, we discuss the all important necessity of properly describing data, as well as touch on several aspects of the language we have not yet covered.

The designer of PASCAL, Niklaus Wirth, entitled one of his books *Data Structures + Algorithms = Programs*, which reflects the importance the description of data has on formulating the algorithms that process that data. If we use an array of chars to represent a character string, for instance, the functions and procedures needed to process them (find their length, concatenate them, and so on) will be vastly different from those required if a linked list is used.

Take, for example, the simple task of finding the length of a string, when this is unknown. Remember that we have padded out any 'spare' array elements of the array representation with ASCII NUL characters, chr (0), or terminated the dynamic list with the pointer value NIL. The array version would seem to be easy enough:

```
FUNCTION Length (S :string):Cardinal;
  VAR
    N       :0..StringLength;
    found   :boolean;
  BEGIN
    N := 0;
    found := false;
    REPEAT
      N := N+1;
      found := S[N]=chr (0)
    UNTIL found OR (N=StringLength);
    IF found
      THEN
        Length := N-1
      ELSE
        Length := StringLength
  END; {Length}
```

TreeSorter Program

This program uses some procedures from the previous CircList program, and these are referenced at the appropriate point. Data consisting of names, the amounts of debt owing and any other fields you care to add are entered from the keyboard and inserted in ascending alphabetic order into a 'binary tree'.

Each node of the tree has two pointers for linking to 'lesser' or 'greater' items. Traversal of the tree is accomplished by comparing the new data with the Name field in each node and taking the appropriate link field (LoBranch or HighBranch). When we find an empty node (that is, one with a NIL value branch) the item is inserted. Writing the data to the file is then achieved simply and naturally by means of a recursive procedure.

You could use these examples as a basis for a very neat database manager — a suite of programs, perhaps, tailored to your own use. Just one word of caution: if you read a sorted file into a binary tree, every insertion will be on the same branch and the tree will become a singly-linked list.

For simpler applications, the circular list is probably the most versatile. There are no NIL pointers in it whatsoever, thus sparing us a double test on each comparison, whereas the binary tree will have (Number of Nodes+1) NILs.

```
PROGRAM   TreeSorter  ( input, output, DataFile );

CONST
        FileName        = 'Treedata';
        StringLength    = 25;

TYPE
   Cardinal     = 0 .. MaxInt;
   StringSize   = 1 .. StringLength;
   string       = PACKED ARRAY [ StringSize ]
                    OF char;
   thing        = RECORD
                    Name    : string;
                    { other fields . . }
                    debt    : Cardinal
                  END;  { thing }
   FileType     = FILE OF thing;
   tree         = ↑ bough;
                  {** forward reference to : }
   bough        = RECORD
                    item        : thing;
                    LowBranch,
                    HighBranch  : tree
                  END;  { bough }

VAR
   data         : thing;
   DataFile     : FileType;
   trunk        : tree;

INCLUDE 'Utils.src' {** source file containing :
                SkipBlanks, ReadCard and ReadLine
                    - see  CircList  program **}
       {lllllllllllllllllllllllllllllllllllllllll}

PROCEDURE  ReadAmount ( VAR  amount  : Cardinal );
       {** Validate a legal Cardinal value **}
  VAR
       OK       : boolean;
  BEGIN
    REPEAT
      REPEAT
        write ( 'Amount ? ' : 20 );

        IF EoLn ( input ) THEN
          ReadLn ( input );

        SkipBlanks ( input )
        {** prompt again if at end of line **}
      UNTIL NOT EoLn ( input );

      ReadCard ( input, amount, OK );

      IF NOT OK THEN
        WriteLn ( '--- ERROR ---' : 20,
                        ' please reenter' )
    UNTIL OK    {** insist on a valid number **}

  END;  { ReadAmount }
       {lllllllllllllllllllllllllllllllllllllllll}

PROCEDURE       Grow    ( VAR   leaf    : tree;
    {** graft on to tree **}   data    : thing );
  BEGIN
    new ( leaf );

    WITH leaf ↑ DO
      BEGIN
        item := data;
```

There is some room here for both error and confusion. Why do we need the local Boolean variable found? And why must we assign the value N-1 to Length? These are minor irritations, but formulating more complicated algorithms can become very error-prone and confusing if we have to resort to artificial devices like this.

Using 'bit flags' (which we know as Boolean types) is the oldest trick in the programmer's book, but they are often introduced simply for the sake of the computer, not as a natural part of the algorithm. The local variable N must be used, not Length, as this is a function identifier, not a variable. Function identifiers on the right-hand side of a statement like:

```
Length := Length+1
```

would attempt to invoke a recursive call to the function Length itself.

Contrast this with the Length function needed for strings using a linked list. With the dynamic representation, remember that the TYPE definition of string is very different, allowing any length of string to be created. Data descriptions such as these will often be defined recursively.

RECURSION

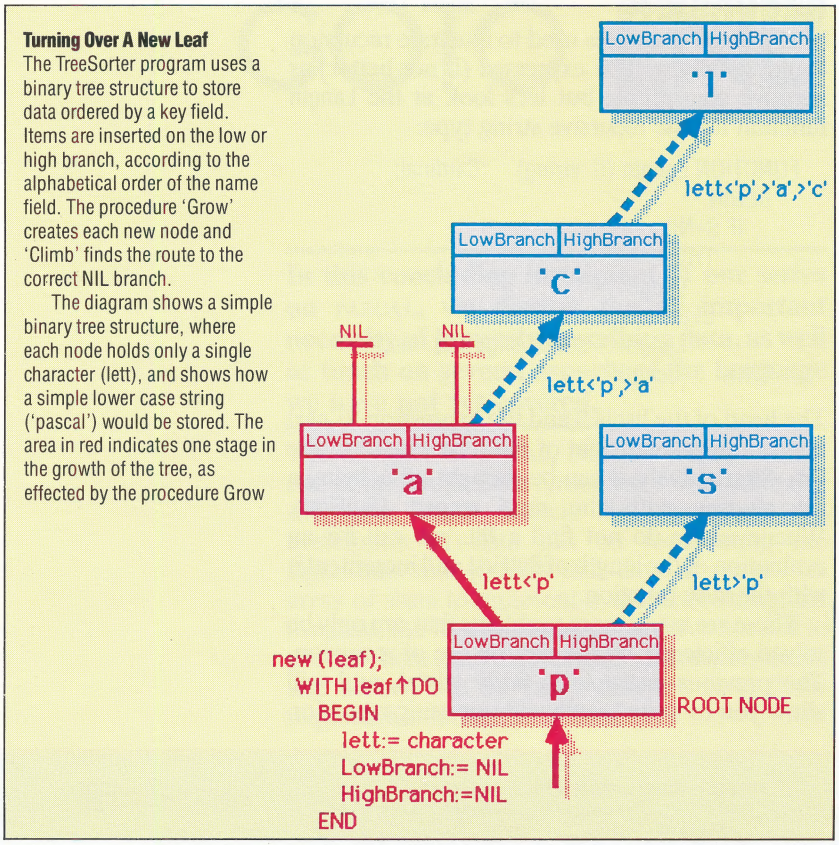
Many trivial examples used to illustrate recursion could equally well be expressed (if not better) as iterative algorithms; but let's look at the Length function for the recursive string type:

```
FUNCTION Length (S :string)    :Cardinal;
  BEGIN
    IF S=NIL
      THEN
        Length := 0
      ELSE
        Length := succ(Length(S↑. next))
  END;   {Length}
```

The head of the list is S and the expression S↑.next selects the pointer field of the next record in the list. Alternatively, it can be thought of as being a list starting with the next record (ButFirst). Whenever we do not find a NIL, we call for an evaluation of the length of ButFirst and increment it with the succ function.

There are many more problems that can only be solved efficiently by the natural use of recursion. The previous example is fairly trivial, and will allow you to write Length without using recursion

```
      LowBranch := NIL; {** No more data yet,  }
      HighBranch := NIL { beyond this bough  **}
    END

  END;  { Grow }
        {lllllllllllllllllllllllllllllllllllllll}

PROCEDURE       Climb          ( shoot    : tree;
  {** find place to insert **}  data     : thing );
  VAR
      stem     : tree;
      larger   : boolean;
  BEGIN
    WHILE shoot <> NIL DO
      BEGIN     {** find an empty bough **}
        stem := shoot;
        larger := data . Name  >
                  stem ↑ . item . Name;
        IF larger
          THEN  {** climb upwards **}
            shoot := shoot ↑ . HighBranch
          ELSE  {** climb downwards **}
            shoot := shoot ↑ . LowBranch
      END;

    Grow ( shoot, data );

    IF larger
      THEN      {** insert above **}
        stem ↑ . HighBranch := shoot
      ELSE      {** insert below **}
        stem ↑ . LowBranch := shoot

  END;  { Climb }
        {lllllllllllllllllllllllllllllllllllllll}

PROCEDURE       Print ( VAR     F       : FileType;
  {** write data to a file **}  root    : tree );
  BEGIN
    IF root <> NIL THEN
      WITH root ↑ DO    {** recursion : }
        BEGIN   {** write out in order : }
          Print ( F, LowBranch ); {** low first,}
          write ( F, item ); { the root itself, }
          Print ( F, HighBranch ) { high last **}
        END

  END;  { Print }
        {lllllllllllllllllllllllllllllllllllllll}

PROCEDURE       Reclaim  ( VAR  root    : tree );
  {** recover allocated memory for further use **}
  BEGIN
    IF root <> NIL THEN {** recurse **}
      BEGIN     {** reclaim branches first - }
        Reclaim ( root ↑ . LowBranch );
        Reclaim ( root ↑ . HighBranch );
        dispose ( root ) { - then the root **}
      END

  END;  { Reclaim }
        {lllllllllllllllllllllllllllllllllllllll}


BEGIN   { TreeSorter - Main program }

  assign ( DataFile, FileName );
  page ( output );
  WriteLn ( '===  Tree Sort  ===' : 25 );
  WriteLn;
  WriteLn ( 'Enter some data (surname first)' );
  write ( 'Name ? ' );
  ReadLine ( data . Name );
  ReadAmount ( data . debt );
  Grow ( trunk, data ); {*'plant the tree trunk *}
  write ( 'Name ? ' );
  ReadLine ( data . Name );

  WHILE data . Name [ 1 ] <> chr ( 0 ) DO
    BEGIN
        ReadAmount ( data . debt );
        Climb ( trunk, data );
        WriteLn ( 'RETURN when done' : 40 );
        write ( 'Name ? ' );
        ReadLine ( data . Name )
    END;
    {** other processing, then write the file : }
  WriteLn ( 'Writing data to : ', FileName );
  rewrite ( DataFile );
  Print ( DataFile, trunk );
  WriteLn;
  WriteLn ( 'Debt' : 8, 'Name' : StringLength );
  WriteLn;
  reset ( DataFile ); {** read the file, }
        { and print the ordered data : }
  WHILE NOT EoF ( DataFile ) DO
    BEGIN       {** read each record **}
        read ( DataFile, data );
          {** write the fields to the VDU : }
        WITH data DO
          WriteLn ( debt : 8, ' : ', Name )
    END;

  Reclaim ( trunk );    {** dispose memory }
    { for other processing etc. **}
END .
```



Turning Over A New Leaf
The TreeSorter program uses a binary tree structure to store data ordered by a key field. Items are inserted on the low or high branch, according to the alphabetical order of the name field. The procedure 'Grow' creates each new node and 'Climb' finds the route to the correct NIL branch.

The diagram shows a simple binary tree structure, where each node holds only a single character (lett), and shows how a simple lower case string ('pascal') would be stored. The area in red indicates one stage in the growth of the tree, as effected by the procedure Grow

LIZ DIXON ON THE MACINTOSH

with little difficulty. The code is larger, and a local counter must be used (as with the array version) specifically to avoid recursion.

```
FUNCTION Length(S :string)    : Cardinal;
  VAR
    N    : Cardinal;
  BEGIN
    N := 0;
    WHILE S <>NIL DO
      BEGIN
        N := N+1;
        S := S↑.next
      END;
    Length := N
  END;  {Length}
```

Even with this iterative algorithm, the clarity and concision are apparent and are due to the simple recursive nature of the data structure.

Many PASCAL compilers support 'directives', which are instructions to the compiler — *not* declarations or statements. The only directive actually required by the ISO Standard is Forward. Should two procedures or functions need to call each other, they are said to be 'mutually recursive'. This occurs only rarely, but it does pose a problem: we can't use any PASCAL object until it has been declared or defined.

The solution is to declare only the heading of one sub-program, replacing its block with the compiler directive FORWARD. After the full definition of the other module, the heading is given in abbreviated form (omitting the parameter list) and the block is defined at that point. Other directives are often available to control compile-time options, but should not be used liberally if you wish to retain portability. This means, besides a special symbol (usually $) having to appear as the first character in the comment, non-portable options may not be flagged by a different compiler.

MORE FLAVOURS

A few non-standard implementations (notably HiSoft) will require slightly different syntax for forward pointer declarations. In this instance, you should refer to your manual. There are very few 'flavours' of this sort, and none whatsoever with PASCAL compilers that conform to the ISO definition of the language.

One other difference that you will find with UCSD, TCL/RML/Oxford and HiSoft versions is the lack of a dispose procedure. In lieu of this, the non-standard procedures mark and release are provided.

There is one more important advanced data description in PASCAL that we have not mentioned – the 'variant'. When we have wanted to store items of different types, we have used a record with appropriately typed fields. But supposing the description of a part or even the whole record needed to be flexible? Typically, we may wish to store different personal information about so-called 'data subjects' depending on, say, whether or not they were married. A variant record has its 'fixed part' defined first, then the variant part is specified by introducing a variant selector (of any simple type) and using the reserved words CASE and OR. For instance:

```
TYPE
  gender = (male, female);
  variant = RECORD
    {any common fields}
    CASE married : boolean OF
      false :( );
      true  : (DateWed :string;
          CASE sex : gender OF
            male : ( );
            female : (MaidenName :string))
    END: {variant}
```

Notice that empty field lists must still have brackets. Space on a file will be fixed (for the largest variant), but memory may be saved with pointers to variants, such as new (p, true, male).

The few weaknesses in PASCAL have been largely eliminated by the efforts of PASCAL experts (and by Wirth in MODULA-2). The language is tremendously powerful and yet small, efficient, general-purpose and easily learnt. Of course, if you don't stray outside the ISO definition, there will be some system-level operations that you cannot implement. But writing these routines in assembler or BCPL and linking them to a PASCAL program is just one possible solution. The benefits of having all your PASCAL source code portable across any micro, mini or mainframe in the world are irresistible. PASCAL is the closest approach we have to a computing *lingua franca.*

COMPETITIVE COMPATIBLE

CHRIS STEVENS

Taking On The Giant
The Tandy 1000 is an IBM PC-compatible machine, which the company hopes will regain some of the market share that has been taken from it by the intervention of the huge IBM Corporation. To achieve compatibility, the Tandy 1000 is based around the Intel 8088 processor and uses the standard 5¼in disk drives. Shown here is the twin disk drive model, which has an additional 128 Kbytes of memory. The computer is displaying Microsoft's Flight Simulator, written for the IBM PC. The ability to run this program is one of the tests of a compatible machine

One of the first entrants into the micro market, the Tandy Corporation has not really achieved any mass appeal, either in the home or business markets. With the introduction of the Tandy 1000, however, the company hopes to provide a relatively inexpensive alternative to the IBM PC and stake its claim in the business market.

Although the Tandy Corporation was one of the first manufacturers to enter into the microcomputer market with the TRS-80, the company failed to gain the international mass appeal of either Commodore in the home market or Apple in the business market — both of whom were early competitors. Tandy now appears to have gone through a regrouping exercise and is poised to make a bilateral attack on the business market. On the one hand, in partnership with ACT (the manufacturers of the Apricot), Tandy has updated its range of machines in its chain of computer centres throughout Europe, which are now selling the Apricot range. The other spearhead in the company's offensive is represented by its own manufacturing division.

It has been obvious to many in the industry for some time that whoever could make a truly IBM PC-compatible machine at a low price would be on to a winner, both in the business market and in the home computer market in the US, where consumers tend to buy machines for home use that are considered too expensive for anything but business applications in Europe.

The Tandy 1000 claims to be a fully compatible machine which, at the entry level of a single disk drive and 128 Kbytes of memory, costs just over £1,100. Tandy doubtless hopes that, by drastically undercutting the opposition, it will be able to revitalise its computer products division.

The computer we shall examine here is the twin-drive 256 Kbyte version of the Tandy 1000. At first sight, it bears more than a passing resemblance to the IBM PC. The machine consists of a large single box containing the computer itself, the interfaces and the disk drives. A monitor rests on top and there is a movable keyboard that can be positioned for maximum convenience.

The Tandy machine has the same solid and reliable appearance as the IBM PC. Like the IBM machine, the Tandy 1000 incorporates a detachable keyboard having sculptured keys with curved layering for maximum ease when typing. Furthermore, to assist optimum positioning, there are two legs underneath that can tilt the keyboard to an angle of 15°.

Although the Tandy has the necessary keys required to achieve compatibility, they are laid out differently from the IBM PC keyboard. This strategy has its advantages and disadvantages. Positively, IBM's keyboard, although praised for its design, has been roundly criticised in many quarters for a certain awkwardness in the positioning of some vital keys, such as the Shift and Alternate keys. The relatively small size of the Return and Control keys has also fuelled the criticism.

Tandy had obviously noted these comments when designing the keyboard. The '/' key, which caused a lot of problems when using the left-hand Shift key, has now been removed altogether — its functions being taken over by SHIFTing keys 4 and

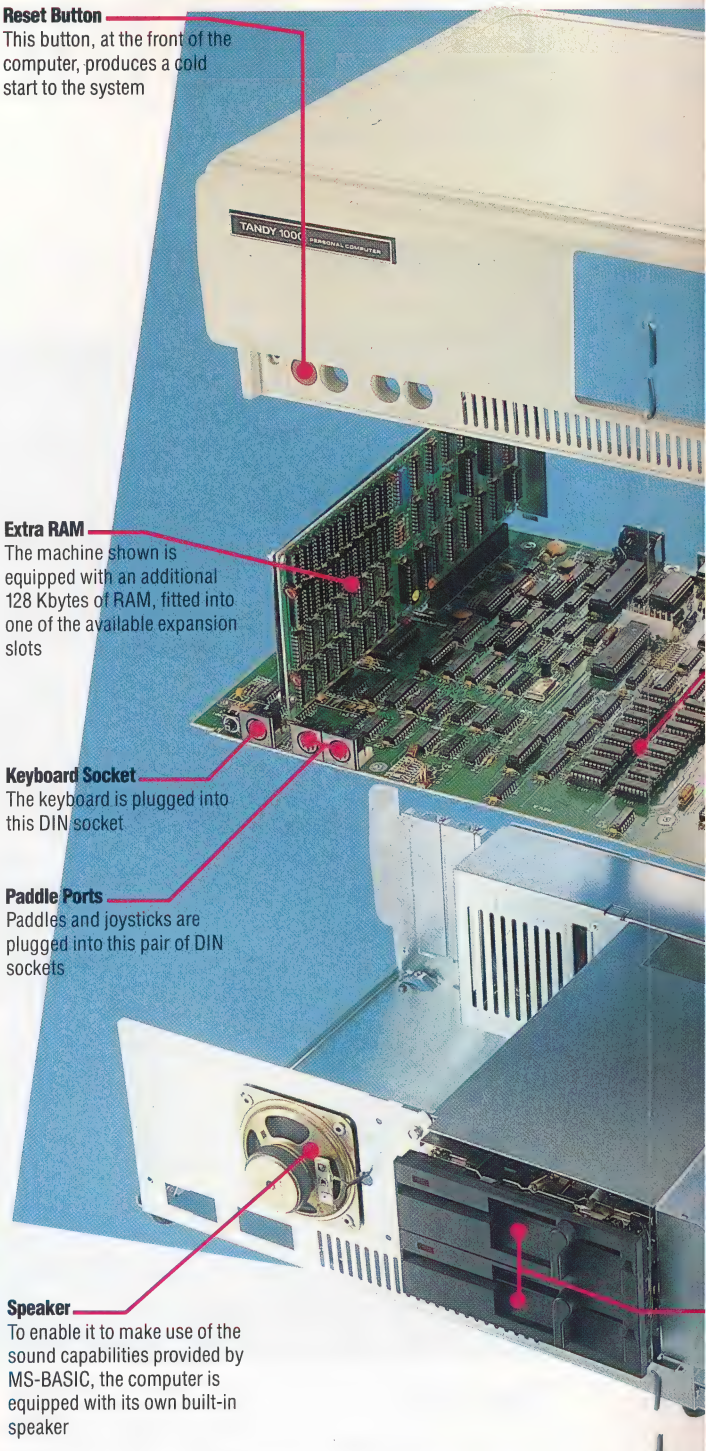
7 on the numeric keypad. Similarly, the Alternate and Caps Lock keys have been moved from their positions on either side of the space bar to the number keypad area and to the far left below the Control keys, respectively. The overall effect of these changes has been to make ordinary typing on the keyboard much easier than on the IBM equivalent.

Of course, the disadvantage to these changes in format is that once you have taken the trouble to familiarise yourself with the IBM PC's idiosyncratic layout, you will have to learn the keyboard again. Newcomers, however, should find the keyboard much easier to get used to. The keys themselves are full travel and feel to be some of the best on the market.

Tandy has also made other changes to the keyboard. The Insert and Delete keys have been incorporated with the + and — functions and are now above the keypad instead of below it. Additional keys include individual cursor keys for editing, as well as Hold, Print and Break keys, all of which have been placed between the typewriter keys and the calculator keypad. The addition of these keys has meant the relocation of the Function keys, from the far left of the keyboard on the IBM PC, to just above the typewriter keys. Although probably unavoidable, the keys are now less conveniently placed than they were, but as compensation, Tandy has provided two additional Function keys.

Turning to the computer itself, Tandy has opted for the large box shape favoured by IBM. On the front, at the left-hand side, are a pair of $5\frac{1}{4}$ in floppy disk drives, although the standard model has only a single drive but with the facility to accommodate another. Although somewhat quieter on average than their IBM counterparts, the drives showed small differences in access speed. One minor irritation is that the disks are not spring loaded. This means they will not pop out of the drives when unlocked and you will have to remove them with your fingers.

Below the front projection next to an air vent is a pair of six-pin 270° DIN plugs for joysticks or other external control devices. To the left of these is a large orange button, the system reset, that



Reset Button
This button, at the front of the computer, produces a cold start to the system

Extra RAM
The machine shown is equipped with an additional 128 Kbytes of RAM, fitted into one of the available expansion slots

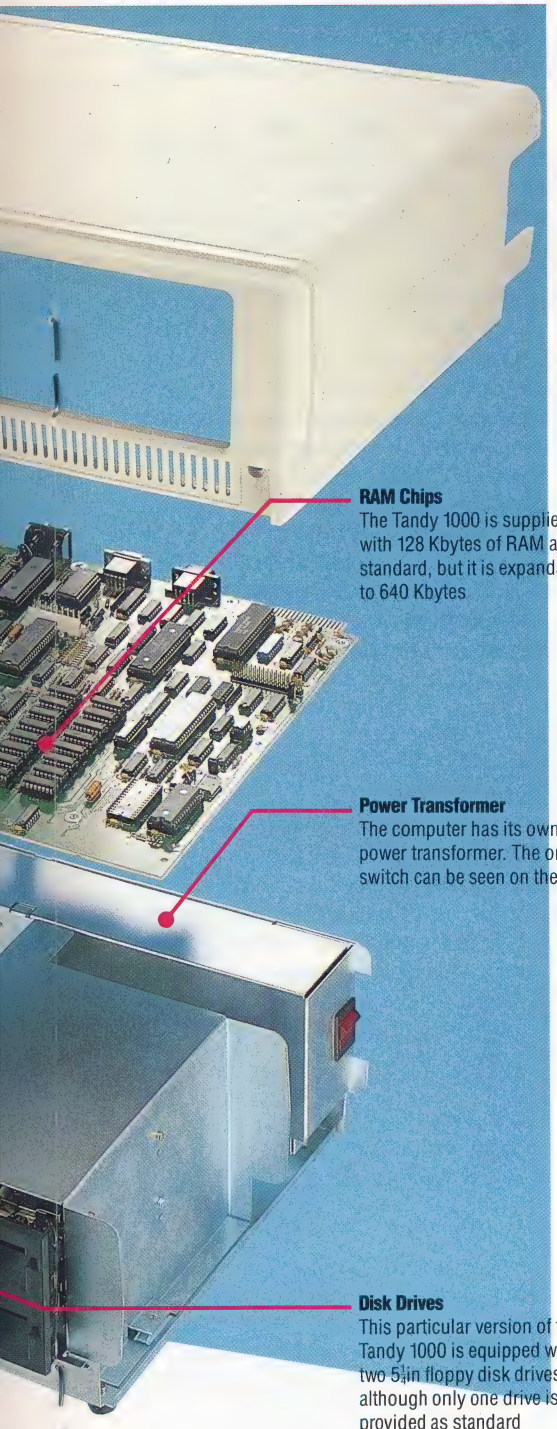
Keyboard Socket
The keyboard is plugged into this DIN socket

Paddle Ports
Paddles and joysticks are plugged into this pair of DIN sockets

Speaker
To enable it to make use of the sound capabilities provided by MS-BASIC, the computer is equipped with its own built-in speaker

Bottoms Up
Apart from having a Centronics printer interface, the Tandy 1000 has both types of monitor socket — RGB and composite video. In order to make the best use of the Tandy's sound capabilities there is also an audio jack provided to amplify the sound through a conventional hi-fi system

CHRIS STEVENS


RAM Chips
The Tandy 1000 is supplied with 128 Kbytes of RAM as standard, but it is expandable to 640 Kbytes
Power Transformer
The computer has its own power transformer. The on/off switch can be seen on the side
Disk Drives
This particular version of the Tandy 1000 is equipped with two 5¼in floppy disk drives, although only one drive is provided as standard

provides a cold start. Despite the fact that it is so prominently displayed, there is little chance of it being pressed inadvertently, since it is set in the casing. In fact, it is probably in the best possible position because a number of applications will not allow you to exit to the operating system without this button being pressed. A conveniently placed reset is therefore helpful, especially on such a large machine.

On the back of the computer are the peripheral interfaces. On the far left of the machine is the power supply input, which is just above the Centronics parallel interface, fitted as an edge connector. To the right is a seven-pin D connector for a light pen and beyond that a seven-socket port for the connection of a RGB monitor. Further to the right, beneath the cooling fan, is a pair of microjack sockets, one of which provides a composite video socket for monochrome or composite colour video monitors; the other is an audio socket that can amplify the computer's sound through a conventional hi-fi system. On the far right are three expansion sockets into which additional peripheral interfaces or memory boards can be added.

COMPARATIVE MERITS

Tandy has emphasised the fact that its machine provides many interface facilities unavailable on the standard IBM PC. Although this is true, the company seems to have provided only the bare minimum necessary for a usable business machine. For example, the entry price machine is fitted with 128 Kbytes of RAM, which is not sufficient to run some of the newer integrated packages. And although the Tandy 1000 is an operational stand-alone machine, an RS232 port (which is optional) is essential for any communications the business user may require. On the other hand, it is certainly better value than the IBM PC itself, which would cost a great deal more for the same facilities. This is, of course, providing it is truly compatible with IBM software.

The problem with attempting to be compatible with the IBM PC is not due to the 8088 CPU or even the MS-DOS operating system, since both can be purchased by approaching the relevant manufacturers. The difficulty lies in the BIOS (Basic Input/Output System), which is IBM's copyright. Many programs use jumps directly to the BIOS routines, and unless a compatible machine has the relevant routines at exactly the same addresses, the program will not run correctly.

In this respect, the Tandy 1000 BIOS, written by Phoenix Compatibility Corporation, is exceptional. Not only does the notoriously difficult Lotus 1-2-3 run on the machine (although even with 256 Kbytes of memory space it was very tight), the computer also runs Wordstar 2000 and dBase II.

Despite this, the Tandy 1000, in common with the IBM PC, is extremely slow at maths computations in BASIC. A zero to 1000 count, used as a simple benchtest, took a full six seconds to complete — slower than many eight-bit micros.

The Tandy 1000 is undoubtedly a viable alternative for the business user wishing to take advantage of the mountain of quality software that has been written for the IBM PC. However, due to hardware limitations imposed by IBM compatibility and the need to keep the price as low as possible, Tandy has come up with a machine that is essentially old-fashioned in business computer terms. Nevertheless, providing IBM does not cut the cost of its machine in retaliation (which seems unlikely), the Tandy Corporation has produced a machine which should prove very successful.

TANDY 1000

PRICE
£1,265 for 128 Kbyte single drive version. Extra disk drive: £269; extra 128 kbyte memory: £249. Monochrome monitor: £184; RGB Monitor: £286

DIMENSIONS
420×335×150mm

CPU
8088, running at 4.77 MHz

MEMORY
128 Kbytes of RAM as standard; expandable to 640 Kbytes

SCREEN
80×25 characters in text mode, 640×200 pixels in high resolution. The machine can display up to eight colours from a choice of 16

INTERFACES
Composite and RGB monitor sockets, printer port, audio jack and three expansion interfaces

LANGUAGES AVAILABLE
BASIC, plus a selection available under MS-DOS, including COBOL, FORTRAN etc.

KEYBOARD
Total of 90 keys, including numeric keypad and 12 function keys

DOCUMENTATION
Documentation includes explanation of MS-DOS, MS-BASIC and the Deskmate integrated package supplied with the machine

STRENGTHS
Fully IBM PC-compatible with a number of interfaces provided which are not available on the basic PC model

WEAKNESSES
Tandy has not advanced the design of the machine much beyond that of the original IBM PC. The use of an 8088 processor rather than the more advanced 8086 means that the Tandy 1000 operates slower than it could

PARAMETER

In computing, a *parameter* is defined as a value that is passed to a subroutine, procedure or function, and will typically be at the beginning of a procedure. This means that when the procedure or subroutine is called by the main program, the parameter will be called with it. This is known as a 'formal parameter' and will be contained within a 'parameter list'.

For example, a typical procedure in PASCAL or LOGO will begin with its identifier, followed by a parameter list enclosed in brackets:

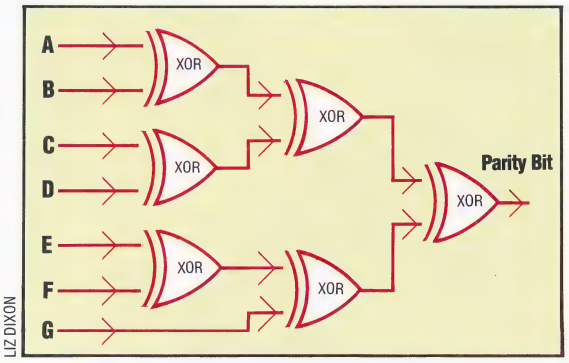
```
PROCEDURE Procname (parameter1, parameter2)
```

Note that in PASCAL, the parameter list will also require the type of number (integer or real) used in the parameter to be listed as well. Naturally, parameters will require data to be provided so that actual values can be assigned to the formal parameter. This data passed to the formal parameter is known as the 'actual parameter'.

Parameters are transferred to a procedure by the technique of 'parameter passing'. Depending on the type of parameter, the actual parameter value is either passed directly or, where the value is variable, the address location is passed to the subroutine.

PARITY

Parity is a method of providing a checksum of a binary pattern and is used to ensure that data received by a computer is correct. 'Parity checking' works on the principle that a binary pattern of, for example, eight bits, will have either an even or odd number of ones. The 'parity bit' is generated through either hardware or software by counting the number of ones in the pattern. If the transmitted number has an even number of ones, the parity bit will be zero and conversely, if the number of ones is odd, then a parity bit of one will be produced.



LIZ DIXON

Just Checking Up
Parity checking is often performed by hardware. In this diagram of an electronic circuit, all the gates are Exclusive-OR (which gives an output of logical one only if the two input lines have different logical values). If the input lines are both zero or both one, the output will be zero. If there is an odd one left over at the final gate, then the parity bit generated at the end will be one; otherwise, the parity bit will be zero

The parity bit is then added to the number of ones. Thus, when the number of ones in the bit pattern plus the parity bit are added together, there should always be an even number of ones. If there is not an even number of ones, a mistake in the transmission has occurred and the computer will ask for the bit pattern to be re-transmitted. This system is known as 'even parity' and the process of verification is called a 'parity check'.

However, even parity is not the only method used. Often, hardware considerations make it necessary for there to be an odd number of ones in the parity checking result. This inverse system is known as 'odd parity'.

Parity checking is most often used in communications where it is important to check incoming data, because the telephone wires used for transmission will often have a great deal of 'noise' (see page 1149) on the wire and they tend to be an unreliable medium. Thus, a method of checking each incoming byte of data for errors is essential.

PASCAL

Named after Blaise Pascal, the inventor of the first four-function mechanical calculator in the 17th century, PASCAL has rapidly become one of the most popular computer languages. The language was created by Niklaus Wirth of the ETH Technical Institute of Zurich at the beginning of the 1970s. It was intended to be used as a teaching aid that would instil in students the ideas of structured programming and quickly became the standard language for teaching computer science. The language also became widely used in commercial applications.

PASCAL has become very popular among professional microcomputer programmers for a number of reasons: its compiler uses a relatively small amount of memory, it is highly structured and its small number of commands can produce a wide number of applications. Furthermore, PASCAL's programs run much faster than their BASIC equivalents because they are compiled rather than interpreted.

Like many other languages, PASCAL now claims a large number of different versions, but there are basically two main forms. The standard version of the language is known as ISO-PASCAL. However, UCSD PASCAL, an enhanced derivative of the standard version, is also gaining a great deal of popularity.

PASS

Like many other terms in computing, *pass* has come to have two very different meanings. In hardware terminology, a pass refers to an entire tape, or the part which holds information, moving past the read/write head of the machine.

The other definition of pass is used when we are assembling or compiling a program. Assembling a program requires two passes; on the first, the assembler will examine the program to set up a symbol table for any labels that we have defined. Because of a process known as 'forward referencing', when the assembler comes across references to a label (normally a branch instruction) that it has not yet encountered, it adds it to the symbol table, before calculating how many bytes will be needed to store the intervening program and assigning addresses to the labels where necessary. On the second pass, the labels can be inserted into their addresses and the program assembled.

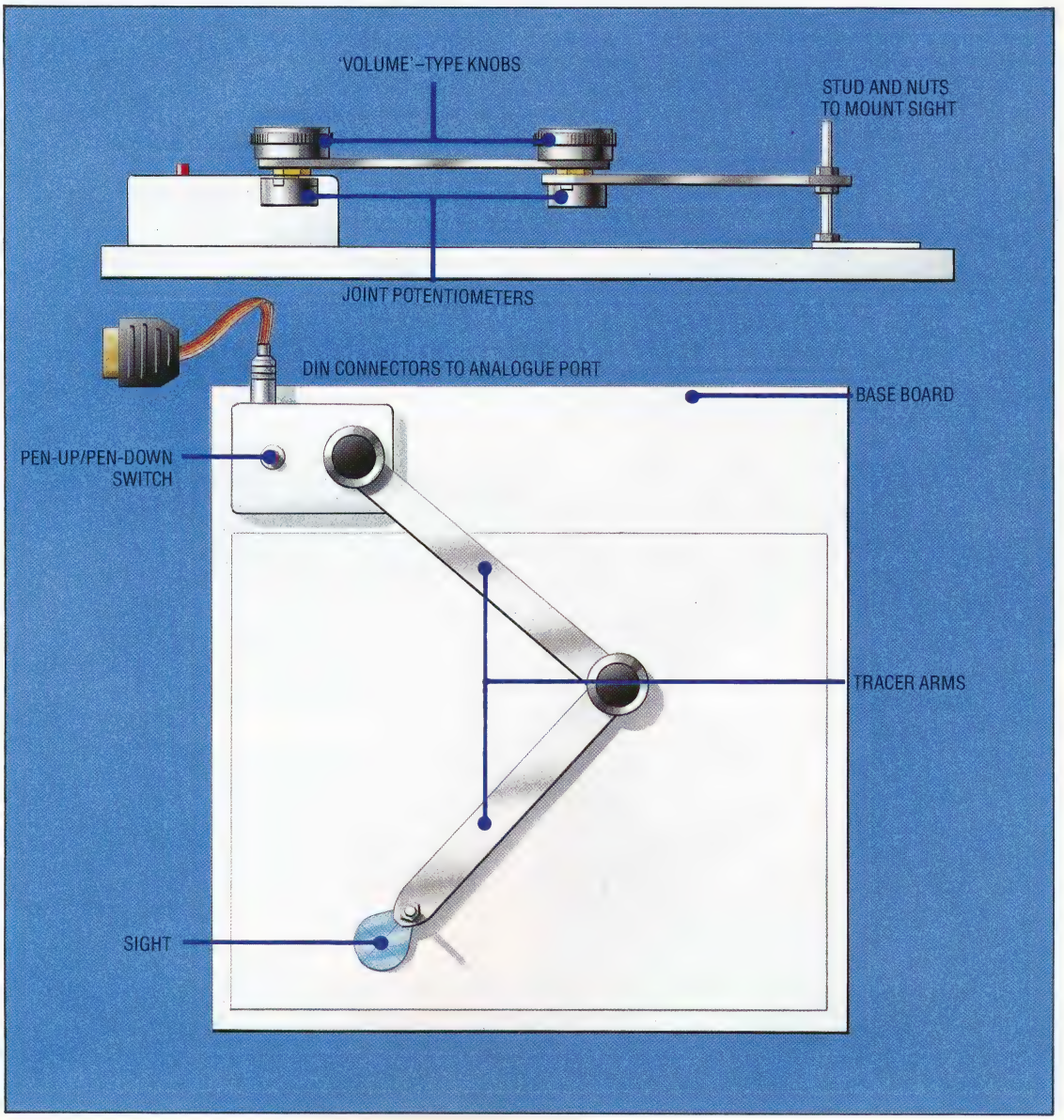
COPY WRITER

We begin a project to build a digital tracer to use with the BBC Micro. Here, we outline the stages in the construction, provide a parts list and begin to put the device together. In future instalments, we will develop the software to control the tracer.

A digital tracer is a simple device that enables you to trace real shapes and figures from a board and display them on a computer screen. Once in digital form, a picture can be edited or saved to disk or tape. The basic component of the device is a metal arm, divided in two and hinged to form a shoulder and elbow joint. Mounted on these hinges are two potentiometers that can be used to provide data about the angles at each joint.

By using the BBC Micro's on-board analogue-to-digital converter, the analogue output from each potentiometer can be converted to digital form and, after calibration, the incoming data can then be manipulated mathematically to give the position of the tip of the second arm relative to the shoulder joint.

The figures illustrated show how the basic constructional elements fit together. The unit is based around a 46cm (18in) square board on which the figure to be traced can be placed. The metal arms are mounted on a small plastic box, on which is also mounted a 5-pin DIN socket that will



KEVIN JONES

Trace Elements
The Workshop digital tracer uses two potentiometers to provide information that can be converted by software, via the BBC Micro's analogue port, to give the position of the sight on the board

accept a connecting cable to the computer's analogue port, as well as a push-to-make switch that provides the tracer with an on-board pen-up/pen-down option.

On the tip of the tracer a small piece of transparent plastic is affixed, with cross-hair markings etched on its surface to form a sight. The mounting system is designed so that the sight can be raised or lowered to facilitate accurate tracing over materials of differing thicknesses.

Parts List

No	Item	Maplin Number
2	100 K-ohm linear potentiometers	FW05F
2	40mm knobs	HB33L
1	Push-to-make switch	FH91Y
1	114×76×38mm box	LF01B
1	5-pin DIN plug	HH28F
1	5-pin DIN socket	HH35Q
1m	10-way ribbon cable	XR06G
1	Pack M5 bolts	BF32K
1	Pack M3 bolts	BF36P
1	Pack M3 nuts	BF58N
1	Pack self-tapping screws	BF69A
1	15-way D plug	BK58N
1	15-way D cover	BK60Q

Miscellaneous

1	600×25×2.5mm aluminium strip*
1	46×46mm melamine covered board**
1	Small piece of transparent plastic***

Notes

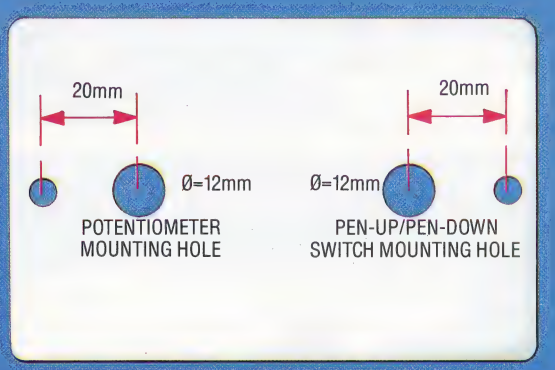
* Available from most DIY and hardware stores. Wooden strips could be used as an alternative

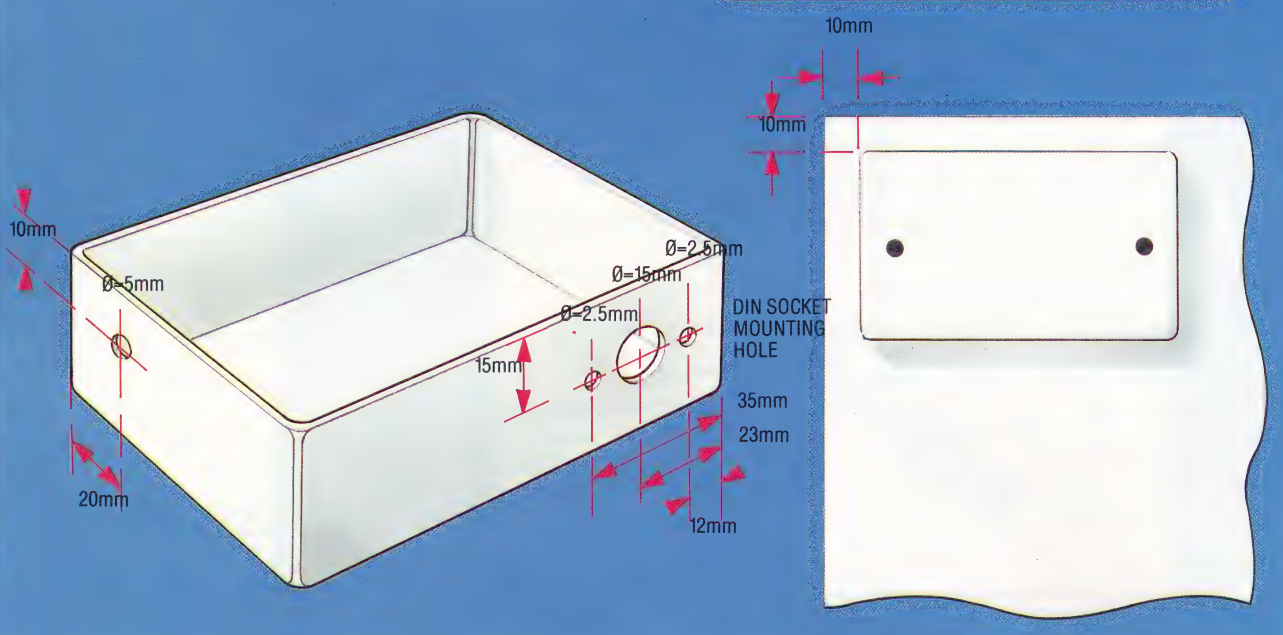
** Often called 'Contiplas'; also available from DIY and hardware stores in various widths and lengths

*** The transparent section from a cassette library case can be used for this

Step 1: Drilling The Box

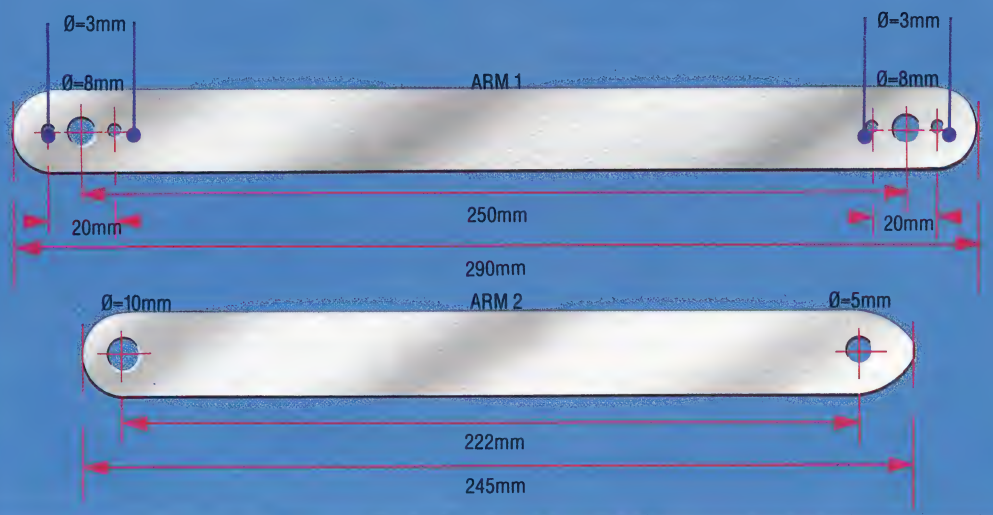
The first stage in the construction of the tracer is, if necessary, to cut the base board to size. The small plastic box should then have a series of holes drilled, into which the tracer arm, switch and DIN socket will be fitted, as shown. When all these holes have been drilled, the box can be mounted at the top left-hand corner of the base board, leaving a 10mm gap around the edge.





Step 2: Cutting The Arm

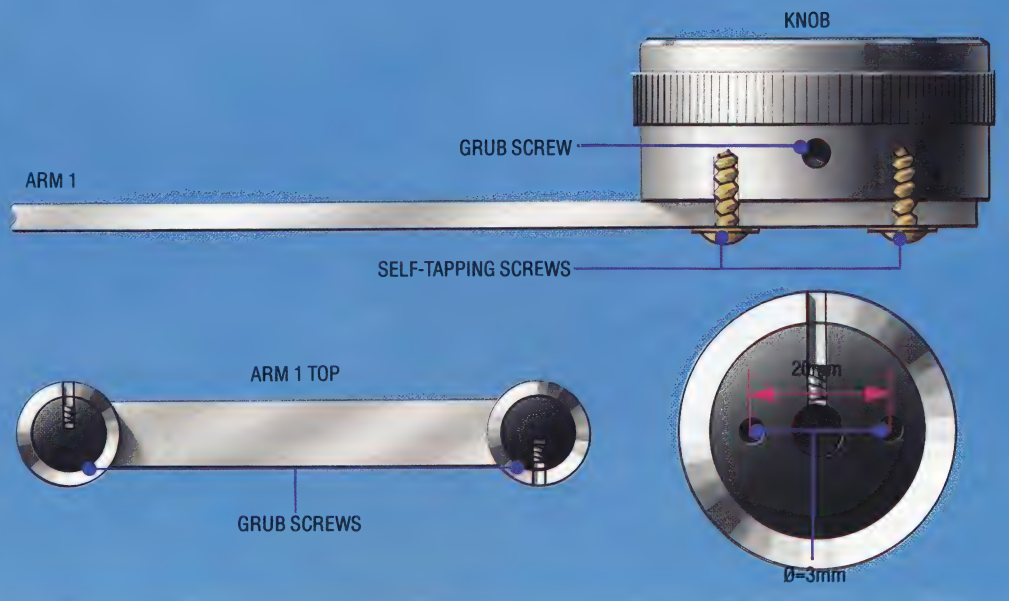
Next, cut the metal strip to form the two lengths shown and drill the required holes. Note, however, that although the exact lengths are not critical, the distances between the holes drilled into each end of the arms must be accurate. This is because these distances will be used by software when making the various geometric calculations necessary to determine the position of the tracer sight when the unit is in use. It is therefore important to measure these distances carefully before you begin to drill.



Step 3: Fastening The Knobs

The final task in this instalment is to mount a 'volume' knob on each end of Arm 1. These knobs will be used to grip the potentiometers that will be implemented in the next instalment of Workshop. Note that the potentiometer spindle is gripped by a small grub screw in the side of the knob. Care should be taken to mount the knobs so that the grub screws are pointing in opposite directions, as shown in the diagram.

The knobs are fitted onto the arm by drilling two 3mm diameter holes into the base of each knob, nearly all the way through. The knobs should then be fastened to Arm 1 with the correct orientation, using self-tapping screws, as shown. In the next instalment, we will complete the construction of the tracer and wire it up to be used with the BBC Micro.



KEVIN JONES

FRAME WORK

This is the first of two instalments in which we explore some of the BASIC interpreter floating point routines for the Commodore 64. In order to demonstrate the use of these routines in programming, we develop a graphics package to rotate a 3-D wire frame figure on the high resolution screen.

The BASIC interpreter floating point routines are not very well documented — for example, there is no jump table of call addresses, as is the case with the kernel — so finding the calls that you will need can be a slow, experimental task. Therefore, to give our explorations a focus, we will aim to produce a machine code graphics routine that will allow us to rotate a 3-D wire frame figure on the high resolution screen.

The mathematical techniques we discuss here could, of course, be used as the basis for other fast arithmetic routines, such as matrix multiplication. It is well to remember that doing arithmetical calculations in real time, in order to create successive screen images, is not necessarily the best programming approach. Often it is better to compute the data first, from which the successive figures will be displayed, before beginning the animation sequence. However, the real-time technique serves our purposes more efficiently.

BASIC variables are stored in memory above the BASIC program. These variables are held in a Variable Table, the start of which is pointed to by the contents of locations 45 and 46 (decimal). Like all pointers in the Commodore 64, these are stored in lo-hi byte format. Therefore, the start address of the Variable Table is pointed to by the formula:

```
PEEK(45)+256*PEEK(46)
```

The various pointers associated with the variables and their normal contents are shown in the following table:

Pointer	Function	Normal Contents
43/44	Start of BASIC	2049
45/46	Start of Variables	Depends on program length
47/48	Start of Arrays	Depends on number of variables
49/50	End of Arrays + 1	Depends on number/size of arrays
51/52	Current bottom strings	Depends on number/length of strings
55/56	Top of memory	40960

Note that dynamic strings build down from the top of memory as they are defined. Arrays, however, are stored above the other variables in the Variable Table and when a new variable is encountered during program execution, the operating system shifts the entire array storage area up by the number of bytes needed to store the new variable.

The storage of strings is necessarily more complicated than that of other variables. Integer variables (not in an array) and floating point variables both require seven bytes, but a string may need up to 255 bytes. To cope with this complication, the operating system only stores the length of the string, and a pointer to its start address, in the Variable Table. If a string is defined in a BASIC program (A$="ABCD", for example) then this pointer will be to the first byte in the string within the BASIC program area. Such a string is called 'static'. If and when the program alters the value of a string, it is then said to be 'dynamic'. The values of dynamic strings are built down from the top of memory and the pointer in the Variable Table altered accordingly. In this way, each entry is held to a constant seven bytes.

The address of a variable in the Variable Table can be easily found from BASIC using a little sleuth work. The secret lies in the fact that just after a variable is called by the BASIC interpreter, its address will be pointed to by the zero page pointer in decimal addresses 71 and 72, and this can be inspected to discover the address. However, we have to be quick to recover this information, since location 71 is also used by the operating system when evaluating some numeric expressions.

The following programs illustrate the format of ordinary variables as they are stored in memory. The first routine recovers a string from memory. Here, we use the technique we have just described to inspect the contents of locations 71 and 72, and immediately store these values in two spare locations in the cassette buffer. These will be used to calculate the address of the string later, in order to recover it.

```
1000 REM**FIND A STRING IN MEMORY**
1010 X$="ABCDEF"
1020 REM**MAKE X$ CURRENT VARIABLE**
1030 X$=X$+""
1040 REM**SAVE POINTER TO VAR TABLE**
1050 POKE828,PEEK(71):POKE829,PEEK(72)
1060 REM**ADDRESS IN VAR TABLE**
1070 ADR=PEEK(828)+256*PEEK(829)
1080 REM**LOOK AT ENTRY IN VAR TABLE**
1090 LS=PEEK(ADR):REM LENGTH OF STRING
1100 SA=PEEK(ADR+1)+256*PEEK(ADR+2)
1110 REM SA IS START ADDR OF STRING
1120 REM**NOW READ STRING**
1130 FORI=SATOSA+LS
1140 VAR$=VAR$+CHR$(PEEK(I))
1150 NEXT
1160 PRINTVAR$
```

You may have thought that using integer variables (denoted by a per cent sign, such as X%) would save memory and speed up the arithmetic. With the Commodore 64, however, this is not the case. When Commodore BASIC has to perform arithmetic on integers, it converts them to floating point and calls the floating point routines! Thus,

although integer variables can be stored in only two bytes, unless they are stored in an array, they will each be allocated seven bytes of memory space. These extra bytes will simply be ignored in the course of processing. The following routine allows you to locate an integer in memory.

```
1000 REM**FIND AN INTEGER IN MEMORY**
1010 X%=3456
1020 REM**MAKE X% CURRENT VARIABLE**
1030 X%=X%
1040 REM**SAVE POINTER TO VAR TABLE**
1050 POKE828,PEEK(71):POKE829,PEEK(72)
1060 REM**ADDRESS IN VAR TABLE**
1070 ADR=PEEK(828)+256*PEEK(829)
1080 REM**LOOK AT ENTRY IN VAR TABLE**
1090 LO=PEEK(ADR+1):HI=PEEK(ADR)
1100 REM**COMPUTE RESULT**
1110 SIGNBIT=(HIAND128)/128
1120 VAR=LO+256*(HIAND127)-32768*SIGNBIT
1130 PRINT VAR
```

The same technique of address location can also be used for a floating point variable. However, there is a far more economic method. This is based on the fact that when DEF FN is used to define a function of the variable X, then X is used (but its value left unchanged) whenever FN is called.

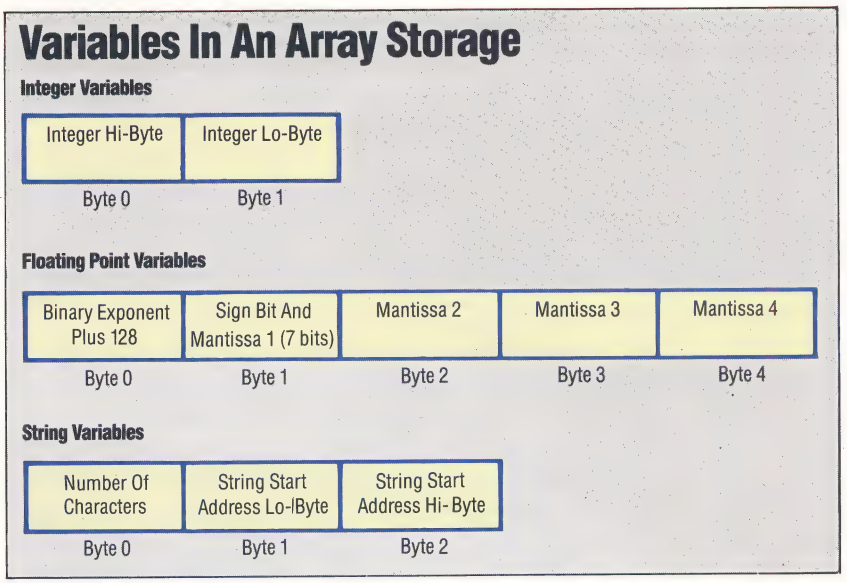
The following program uses DEF FN to calculate the address of the current BASIC variable, held in locations 71 and 72. As X is used as the function variable, this ensures that the address generated is that of the first byte of X held in the BASIC Variable Table. The FN is then called, assigning this address to the variable ADD. Note that passing a zero (or any other value) via the FN command will not change the value of X, held in the Variable Table, or the address calculated by the function.

```
1000 REM**FIND A FPVAR IN MEMORY**
1010 DEF FNADR(X)=PEEK(71)+256*PEEK(72)
1020 ADD=FNADR(0):REM ALWAYS RETURNS ADDRESS OF X
1030 X=-3.14159
1040 REM**CONVERT FROM BASE 2 TO DECIMAL
1050 POWERTWO=2^(PEEK(ADD)-129)
1060 SIGN=(-1)^((PEEK(ADD+1)AND128)/128)
1070 REM*FRACTION PART IS 31 BITS WIDE*
1080 D1=PEEK(ADD+1)AND127:REM 7 BITS
1090 D2=PEEK(ADD+2):REM 8 BITS
1100 D3=PEEK(ADD+3):D4=PEEK(ADD+4)
1110 REM**GULP!**
1120 FRACT=2^(-7)*D1+2^(-15)*D2+2^(-23)*D3+2^
     (-31)*D4
1130 MANT=1+FRACT
1140 VAR=SIGN*POWERTWO*MANT
1150 PRINTVAR
```

ARRAYS IN MEMORY

When a DIMension statement is executed, memory is reserved for the array. This consists of an array header plus the number of bytes needed for element storage. The format of the elements stored in an array is different for each of the variable types we've looked at here. The diagram shows these variations. We will need to understand these differences when accessing array elements from machine code.

Now, let's have a close look at some floating point arithmetic. When the BASIC interpreter is performing floating point calculations, it stores any intermediate results in two 'floating point accumulators'. These are generally called FAC and ARG respectively, and the format used is the same as that used when storing variables in memory. FAC is found at addresses $61 to $65 (decimal 97 to 101), and ARG uses $69 to $6D (decimal 105 to 109). For simplicity's sake, we will continue using interpreter routines that only shuffle numbers between FAC and the memory.



Variable Types
Three array variable types exist on the Commodore 64, each having a different format, when held as bytes in memory. Integer variables are held in two bytes as two's-complement numbers. Floating point numbers require five bytes to hold the mantissa, a sign bit and exponent. String variables are held in a different area of memory, using one byte for each character in the string. However, this data is not held in the array table; instead, three bytes are used to give the number of characters in the string and the 16-bit address that points to the start of the string character data area

The interpreter routines we will become familiar with in this section are as follows:

- MOVFM (call address $BBA2):
The function of this routine is to load the contents of FAC from a floating point variable in memory. This is represented symbolically as: F←M. To call it, load the accumulator with the lo-byte of the start address of the variable in memory, and the Y register with the hi-byte.

- MOVMF (call address $BBD4):
This routine puts the contents of FAC into seven bytes in memory, or M←F. To call it, load the X register with the lo-byte and the Y register with the hi-byte of the start-byte destination of the variable in memory.

- FMULT (call address $BA28):
This is the multiplication routine, which multiplies the contents of FAC with a second variable in memory, and stores the result in FAC. We place the first variable into FAC by using MOVFM and point to the second variable by loading the accumulator with the lo-byte and the Y register with the hi-byte of the start-byte, before calling this routine. Finally, we can, if required, return the result back into memory using MOVMF.

- FADD (call address $B867).
This addition routine performs FAC=MEM-FAC. To call it, we must load the accumulator with the lo-byte of MEM and the Y register with the hi-byte of MEM.

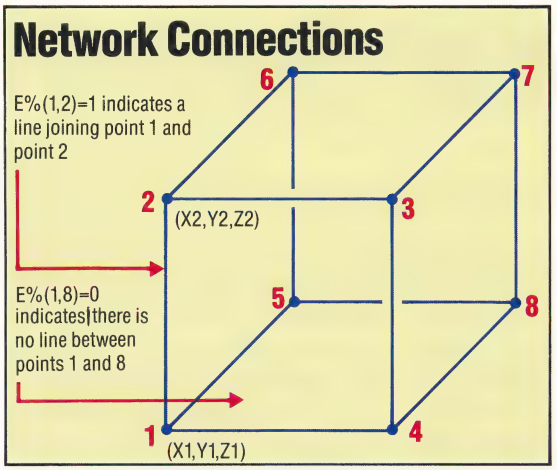
- FSUB (call address $B850):
This subtraction routine performs FAC=MEM-FAC. To call it, we load the accumulator with the lo-byte of MEM and place the hi-byte of MEM in the Y register.

These are the interpreter routines, which we shall build into the first phase of our graphics program. Let's now look at the components of that program.

The underlying idea of this graphics project is that a wire frame figure can be specified by a

All Wired Up

The ROTSUB program rotates a wire-framed figure on the Commodore 64's high resolution screen. This figure is defined using four arrays, three of which are used to hold X, Y and Z co-ordinates of each node. A fourth array — E%(,) — is used to define whether or not two particular nodes are to be connected by a line. In the example, the four arrays are initialised to produce a cube figure as shown



number of points (or nodes) and a matrix of edge connections. The nodes have co-ordinates X(I),Y(I),Z(I) — where I runs from 1 to NP (the number of points). The matrix of edge connections is E%(I,J) — where both I and J run from 1 to NP. We define E%(I,J) as equal to one if the point I is connected to the point J. Otherwise, it is equal to zero. This is, of course, not the most economic use of memory, because we are using two bytes where one will do. But it makes it much easier for the user to specify which points are to be connected. Also, in practical applications, NP will not be very large.

To communicate with the machine code rotation routine from BASIC, we POKE the base addresses of the arrays starting from element 1. Thus, the addresses of X(1),Y(1),Z(1) and E%(1,1) will need to be found. In addition, we will supply the machine code with the number of points, NP, and the COS and SIN of the angle of rotation required about the Z axis.

Any ambitious machine code project had better proceed in careful small steps, or else you will be faced with an enormous debugging task later. For this reason, we have planned the project in three phases. First, we define the algorithms and write a BASIC test program. This is accomplished in the Rotating Cube program given here, which rotates a three-dimensional cube about the Z axis and projects the result onto the X,Y-plane.

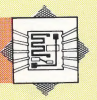
Our eventual goal is to turn the BASIC listing into machine code. But we begin with the less ambitious task of rendering the subroutine at line 1800 of the version into machine code. This is accomplished in the I-Rotsub program, and a test program for this routine is given.

In the next instalment, we will complete the project and add the remainder of the code to the Assembly listing, as well as give a BASIC Loader for the entire program.

I-ROTSUB 64 Program

The following Assembly listing should be assembled and loaded, and the resulting code saved as I-ROT.HEX

```
;++++++++++++++++++++++++++
;++    I-ROTSUB 64       ++
;++                      ++
;++++++++++++++++++++++++++
;++USES ARRAYS DEFINED   ++
;++FROM BASIC ADDRESSES  ++
;++MUST BE POKED FIRST   ++
;++++++++++++++++++++++++++
;
;PLTSUB = $C183
;XLO    = $C103
;XHI    = $C104
;YLO    = $C105
;
*       = $C544
;
;++++ ROTSUB  VARIABLES ++++
;
; VARIABLES CALLED FROM BASIC
XBASLO  *=*+1 ; POKE50500,X(0)LO
XBASHI  *=*+1 ; POKE50501,X(0)HI
YBASLO  *=*+1 ; POKE50502,Y(0)LO
YBASHI  *=*+1 ; POKE50503,Y(0)HI
NP      *=*+1 ; POKE50504,NP
; VARIABLES USED BY M/C
XILO    *=*+1
XIHI    *=*+1
YILO    *=*+1
YIHI    *=*+1
CSLO    *=*+1
CSHI    *=*+1
SNLO    *=*+1
SNHI    *=*+1
MEM1    *=*+5 ;FLOATING POINT VAR
MEM2    *=*+5 ;FLOATING POINT VAR
FAC      =$0061
ARG      =$0069
;
;INTERPRETER ARITHMETIC CALLS
;
FMULT    =$BA28 ; FAC=FAC*ARG
FADDT    =$B86A ; FAC=FAC+ARG
FSUB     =$B850 ; FAC=MEM-FAC
FADD     =$B867 ; FAC=FAC+MEM
MOVFM    =$BBA2 ; FAC=MEMORY
MOVMF    =$BBD4 ; MEMORY=FAC
;
;++++ SAVE REGISTERS++
;
        PHA
        TXA
        PHA
        TYA
        PHA
;
;++++ INITIALISE VARIABLES ++++
;
        LDA XBASLO
        STA XILO
        LDA XBASHI
        STA XIHI
        LDA YBASLO
        STA YILO
        LDA YBASHI
        STA YIHI
;
;++++PERFORM  MEM1=X(I)*CS-Y(I)*SN
;
START
        LDA XILO
        LDY XIHI
        JSR MOVFM   ; FAC = X(I)
        LDA CSLO
        LDY CSHI
        JSR FMULT   ; FAC = X(I)*CS
        LDX #<MEM1
        LDY #>MEM1
        JSR MOVMF   ; MEM1=X(I)*CS
        LDA YILO
        LDY YIHI
        JSR MOVFM   ; FAC = Y(I)
        LDA SNLO
        LDY SNHI
        JSR FMULT   ; FAC = Y(I)*SN
        LDA #<MEM1
        LDY #>MEM1
        JSR FSUB    ; FAC = MEM1-FAC
        LDX #<MEM1
        LDY #>MEM1
        JSR MOVMF   ; MEM1= FAC
;
;++++PERFORM MEM2=Y(I)*CN+X(I)*SN
;
        LDA YILO
        LDY YIHI
        JSR MOVFM   ; FAC = Y(I)
        LDA CSLO
        LDY CSHI
        JSR FMULT   ; FAC = Y(I)*CS
        LDX #<MEM2
        LDY #>MEM2
        JSR MOVMF   ; MEM2= Y(I)*CS
        LDA XILO
        LDY XIHI
        JSR MOVFM   ; FAC = X(I)
        LDA SNLO
        LDY SNHI
        JSR FMULT   ; FAC = X(I)*SN
        LDA #<MEM2
        LDY #>MEM2
        JSR FADD    ; FAC = MEM2+FAC
        LDX #<MEM2
        LDY #>MEM2
        JSR MOVMF   ; MEM2= FAC
;
;++++PERFORM X(I) =MEM1:Y(I)=MEM2
;
        LDA #<MEM1
        LDY #>MEM1
        JSR MOVFM   ; FAC =MEM1
        LDX XILO
        LDY XIHI
        JSR MOVMF   ; X(I)=FAC
        LDA #<MEM2
        LDY #>MEM2
        JSR MOVFM   ; FAC =MEM2
        LDX YILO
        LDY YIHI
        JSR MOVMF   ; Y(I)=FAC
;
;++++TEST END LOOP
;
        DEC NP
        BEQ EXIT
;
;++++INCREMENT ARRAY POINTERS
;
        LDA #$05
        CLC
        ADC XILO
        STA XILO
        BCC XNOHI
        INC XIHI
XNOHI
        LDA #$05
        CLC
        ADC YILO
        STA YILO
        BCC YNOHI
        INC YIHI
YNOHI
        JMP START
;
;++++ PULL REGISTERS OFF STACK ++++
;
EXIT
        PLA
        TAY
        PLA
        TAX
        PLA
;
        RTS
.END
```



BASIC Rotating Cube Program

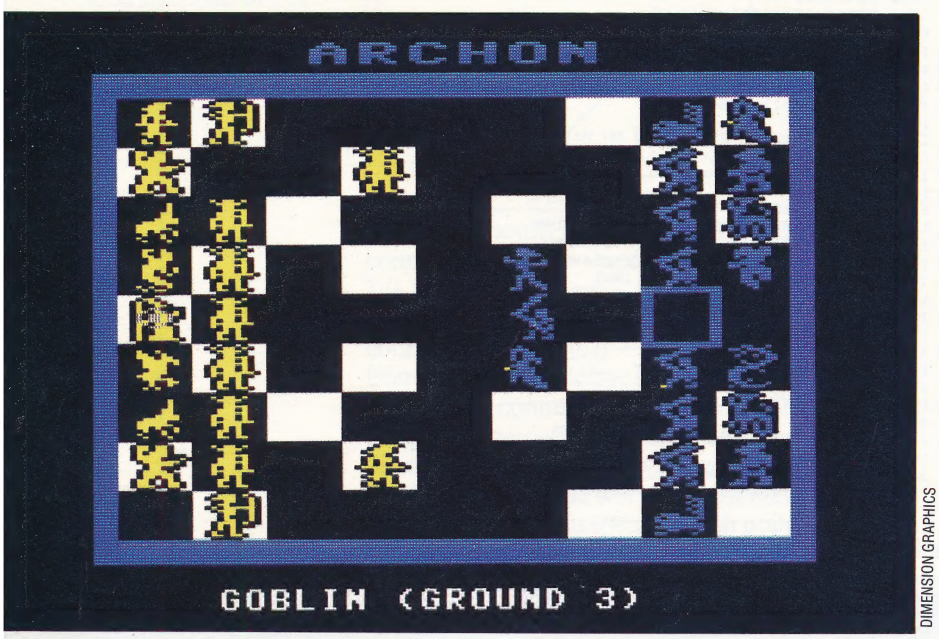
```
1000 REM** BASIC ROTATING CUBE**
1010 IFA=0THENA=1:LOAD"PLOTSUB.HEX",8,1
1020 IFA=1THENA=2:LOAD"LINESUB.HEX",8,1
1030 REM**DIMENSION ARRAYS**
1040 NP=8:REM NUMBER OF POINTS
1050 DIM X(NP),Y(NP),Z(NP)
1060 DIM ED(NP,NP):REM EDGE CONNECTIONS
1070 REM**INITIALISE ARRAYS**
1080 REM--CUBE COORDINATE DATA
1090 DATA  75, 75, 75:REM----------/1
1100 DATA -75, 75, 75:REM TOP FOUR /2
1110 DATA -75,-75, 75:REM POINTS   /3
1120 DATA  75,-75, 75:REM----------/4
1130 DATA  75, 75,-75:REM----------/5
1140 DATA -75, 75,-75:REM BOT FOUR /6
1150 DATA -75,-75,-75:REM POINTS   /7
1160 DATA  75,-75,-75:REM----------/8
1170 REM**ROTATE CUBE ABOUT X-AXIS π/4
1180 FORI=1TONP
1190 READX(I),Y(I),Z(I)
1200 Y(I)=Y(I)*COS(π/4)-Z(I)*SIN(π/4)
1210 Z(I)=Z(I)*COS(π/4)+Y(I)*SIN(π/4)
1220 NEXT
1230 REM**ROTATE CUBE ABOUT Z-AXIS π/4
1240 FORI=1TONP
1250 X(I)=X(I)*COS(π/4)-Y(I)*SIN(π/4)
1260 Y(I)=Y(I)*COS(π/4)+X(I)*SIN(π/4)
1270 NEXT
1280 REM--EDGE CONNECTION DATA--
1290 E(1,2)=1:REM 1 CONNECTED TO 2
1300 E(2,3)=1:E(3,4)=1:E(4,1)=1
1310 E(5,6)=1:REM BOT SQUARE
1320 E(6,7)=1:E(7,8)=1:E(8,5)=5
1330 E(5,1)=1:REM TOP TO BOT EDGES
1340 E(6,2)=1:E(7,3)=1:E(8,4)=1
1350 REM**SYMMETRISE E(I,J)**
1360 FORI=1TONP:FORJ=1TONP
1370 IFE(I,J)=1THENE(J,I)=1
1380 NEXT:NEXT
1390 REM#########################
1400 REM##PLOT ROTATING CUBE##
1410 SA=2*π/45
1420 FOR A=π/4 TO π/4+2*π STEP SA
1430 GOSUB1800:REM ROTATE THRU SA
1440 GOSUB1590:REM INIT/CLEAR SCREEN
1450 REM--PLOT CUBE--
1460 FORI=1TONP
1470 FORJ=1TOI
1480 IF E(I,J)=0THEN1510:REM NOT JOINED
1490 GOSUB1630:REM COMPUTE PROJECTION
1500 GOSUB1670:REM JOIN POINTS
1510 NEXT:NEXT
1520 REM--------------
1530 NEXT A:REM NEXT ANGLE
1540 REM#########################
1550 REM**WAIT**
1560 GETA$:IFA$=""THEN1560
1570 GOSUB1760:REM RESET SCREEN
1580 END
1590 REM**SETUP HIRES**
1600 POKE49408,1:POKE49409,1
1610 POKE49410,1:SYS49422
1620 RETURN
1630 REM**COMPUTE PROJECTION ON HIRES**
1640 X1%=X(I)+159:Y1%=199-(Z(I)+100)
1650 X2%=X(J)+159:Y2%=199-(Z(J)+100)
1660 RETURN
1670 REM**LINESUB**
1680 IF(X1%=X2%)AND(Y1%=Y2%)THENRETURN
1690 MHI=INT(X1%/256):MLO=X1%-256*MHI
1700 NHI=INT(X2%/256):NLO=X2%-256*NHI
1710 POKE49920,MLO:POKE49921,MHI
1720 POKE49922,NLO:POKE49923,NHI
1730 POKE49924,Y1%:POKE49925,Y2%
1740 SYS49934:REM LINESUB
1750 RETURN
1760 REM**RESET SCREEN**
1770 POKE49408,0:SYS49422
1780 PRINTCHR$(147)
1790 RETURN
1800 REM**ROTATE CUBE ABOUT Z-AXIS/SA
1810 FORI=1TONP
1820 X(I)=X(I)*COS(SA)-Y(I)*SIN(SA)
1830 Y(I)=Y(I)*COS(SA)+X(I)*SIN(SA)
1840 NEXT
1850 RETURN
```

Test I-ROTSUB Program

The following BASIC program should be entered and saved as TEST I-ROT. Note that no variable should be defined between executing the code in lines 1880 to 2000 and calling SYS50523. Otherwise, the base address of the arrays will be incorrectly passed to the machine code and the program will crash

```
1000 REM** TEST I-ROT **
1010 IFA=0THENA=1:LOAD"PLOTSUB.HEX",8,1
1020 IFA=1THENA=2:LOAD"LINESUB.HEX",8,1
1030 IFA=2THENA=3:LOAD"I-ROT.HEX",8,1
1040 REM**DIMENSION ARRAYS**
1050 NP=8:REM NUMBER OF POINTS
1060 DIM X(NP),Y(NP),Z(NP)
1070 DIM ED(NP,NP):REM EDGE CONNECTIONS
1080 REM**INITIALISE ARRAYS**
1090 REM--CUBE COORDINATE DATA
1100 DATA  75, 75, 75:REM----------/1
1110 DATA -75, 75, 75:REM TOP FOUR /2
1120 DATA -75,-75, 75:REM POINTS   /3
1130 DATA  75,-75, 75:REM----------/4
1140 DATA  75, 75,-75:REM----------/5
1150 DATA -75, 75,-75:REM BOT FOUR /6
1160 DATA -75,-75,-75:REM POINTS   /7
1170 DATA  75,-75,-75:REM----------/8
1180 REM**ROTATE SPACE ABOUT X-AXIS π/4
1190 FORI=1TONP
1200 READX(I),Y(I),Z(I)
1210 Y(I)=Y(I)*COS(π/4)-Z(I)*SIN(π/4)
1220 Z(I)=Z(I)*COS(π/4)+Y(I)*SIN(π/4)
1230 NEXT
1240 REM**ROTATE SPACE ABOUT Z-AXIS π/4
1250 FORI=1TONP
1260 X(I)=X(I)*COS(π/4)-Y(I)*SIN(π/4)
1270 Y(I)=Y(I)*COS(π/4)+X(I)*SIN(π/4)
1280 NEXT
1290 REM--EDGE CONNECTION DATA--
1300 E(1,2)=1:REM 1 CONNECTED TO 2
1310 E(2,3)=1:E(3,4)=1:E(4,1)=1
1320 E(5,6)=1:REM BOT SQUARE
1330 E(6,7)=1:E(7,8)=1:E(8,5)=5
1340 E(5,1)=1:REM TOP TO BOT EDGES
1350 E(6,2)=1:E(7,3)=1:E(8,4)=1
1360 REM**SYMMETRISE E(I,J)**
1370 FORI=1TONP:FORJ=1TONP
1380 IFE(I,J)=1THENE(J,I)=1
1390 NEXT:NEXT
1400 REM#########################
1410 REM##PLOT ROTATING CUBE##
1420 SA=2*π/45:CS=COS(SA):SN=SIN(SA)
1430 FOR A=0 TO 2*π STEP SA
1440 GOSUB1870:REM ROTATE THRU SA
1450 GOSUB1600:REM INIT/CLEAR SCREEN
1460 REM--PLOT CUBE--
1470 FORI=1TONP
1480 FORJ=1TOI
1490 IF E(I,J)=0THEN1520:REM NOT JOINED
1500 GOSUB1640:REM COMPUTE PROJECTION
1510 GOSUB1680:REM JOIN POINTS
1520 NEXT:NEXT
1530 REM--------------
1540 NEXT A:REM NEXT ANGLE
1550 REM#########################
1560 REM**WAIT**
1570 GETA$:IFA$=""THEN1570
1580 GOSUB1770:REM RESET SCREEN
1590 END
1600 REM**SETUP HIRES**
1610 POKE49408,1:POKE49409,1
1620 POKE49410,1:SYS49422
1630 RETURN
1640 REM**COMPUTE PROJECTION ON HIRES**
1650 X1%=X(I)+159:Y1%=199-(Z(I)+100)
1660 X2%=X(J)+159:Y2%=199-(Z(J)+100)
1670 RETURN
1680 REM**LINESUB**
1690 IF(X1%=X2%)AND(Y1%=Y2%)THENRETURN
1700 MHI=INT(X1%/256):MLO=X1%-256*MHI
1710 NHI=INT(X2%/256):NLO=X2%-256*NHI
1720 POKE49920,MLO:POKE49921,MHI
1730 POKE49922,NLO:POKE49923,NHI
1740 POKE49924,Y1%:POKE49925,Y2%
1750 SYS49934:REM LINESUB
1760 RETURN
1770 REM**RESET SCREEN**
1780 POKE49408,0:SYS49422
1790 PRINTCHR$(147)
1800 RETURN
1810 REM**ROTATE CUBE ABOUT Z-AXIS/SA
1820 FORI=1TONP
1830 X(I)=X(I)*CS-Y(I)*SN
1840 Y(I)=Y(I)*CS+X(I)*SN
1850 NEXT
1860 RETURN
1870 REM**ROTATE ABOUT Z-AXIS/SA
1880 X(1)=X(1):REM X(1) CURRENT VAR
1890 POKE50500,PEEK(71):REM X(1)LO
1900 POKE50501,PEEK(72):REM X(1)HI
1910 Y(1)=Y(1):REM Y(1) CURRENT VAR
1920 POKE50502,PEEK(71):REM Y(1)LO
1930 POKE50503,PEEK(72):REM Y(1)HI
1940 POKE50504,NP:REM NUMBER OF POINTS
1950 CS=CS:REM MAKE CS CURRENT VAR
1960 POKE50509,PEEK(71)
1970 POKE50510,PEEK(72)
1980 SN=SN:REM MAKE SN CURRENT VAR
1990 POKE50511,PEEK(71)
2000 POKE50512,PEEK(72)
2010 SYS50523
2020 RETURN
```

ELEMENTAL FORCES



DIMENSION GRAPHICS

Squaring Up
Archon can best be described as an animated game of chess. When one of the pieces is ordered to move to another square on the board, it will march, wriggle or fly depending on what sort of character it is. Once there, if the square is occupied by a piece of the opposing side then the screen will change to a close up of the square and battle will commence. The outcome of the battle is decided by the skill of the player and the relative strengths of the opposing pieces

Combining elements of chess and the fast moving graphics of arcade games, Archon, from Ariolasoft, enlists the player's skills as both a strategist and adept joystick handler. We look at how the classic board game is successfully married to computer-generated graphics as the forces of Light do battle with Dark enemies.

Archon is a game that can be fascinating to the chess player, yet can also satisfy the most hardened arcade enthusiast. The basis of the game is the battle between Light and Dark forces. The game begins with a display of a strategy board on which the two opposing sides are positioned in rows and columns, in much the same way as chess pieces. The pieces have names such as 'phoenix' and 'knight' on the side of Light, while 'banshees', 'goblins' and 'dragons' distinguish the forces of Dark. The icons representing the pieces on either side have different strength and movement capabilities. Some pieces, for example, can 'fly', which means they can jump over pieces in front of them, whereas others have only 'ground' movement.

The board itself is divided into a nine by nine matrix. At first sight, it looks very similar to a chess board, with many of the squares coloured alternately white and black. Other squares, however, change from black to white and back again, as the game progresses. The reason for this is that the Light forces are stronger on white squares and the Dark are stronger on black.

The object of Archon is to occupy the five 'power points'; four are positioned in a cross on the edges of the board with the crux at the centre square. When the game begins, the opening strategy dictates that you move your icons from their opposite colour, where they are vulnerable to attack, to colours on which they are more powerful. By doing this, you can also open up the back row so that 'ground' pieces are free to move, enabling you to construct an effective barrier to prevent opposing ground forces from disrupting the defence.

At this point, the game appears to have most of the elements of chess transposed onto the computer. However, the real difference in the game becomes evident when attempting to occupy a square already held by an opposing icon. When moving an icon to an enemy-occupied square with the joystick, the screen reveals a close-up of the square, instead of the piece being automatically taken. The enemy pieces are positioned at either side of the square, on which there is a bar representing the icon's strength, and an arcade-style shoot-out begins. As the bar is struck by the opposing piece, this strength will diminish, and when it disappears completely, the opposition wins and occupies the square. As the strengths and methods of attack differ radically from icon to icon, some of these battles are more evenly matched than others. For example, a dragon can shoot flame across the square at an opponent, whereas a knight has to be very close to an enemy icon in order to use a sword. The battles are further complicated by barriers scattered across the square, which change as the fight progresses.

Each side has an icon that can cast a spell: a wizard for Light and a sorceress for Dark. The spells are identical, but they can only be used by the icon once. The spells include such powers as 'reviving' a piece to a square beyond its movement capabilities or summoning a creature known as an 'elemental'. These can be conjured to attack an enemy piece and are useful when an enemy piece threatens to attack a much weaker piece of yours, or if you want to weaken your opponent's. However, even if the elemental destroys the enemy icon it disappears at the end of the battle. An elemental cannot occupy a square.

The game can be played against either the computer or another person. Although the computer is merely an average strategist, it proves to be a formidable opponent in the arcade section and beginners may find themselves hard pressed to win even a single battle in the first few plays.

Archon: For the Commodore 64
Price: £9.95 (inc VAT)
Publishers: Ariolasoft, Dept A.S. 72, Westfields Avenue, London, SW13 0AU
Authors: Anne Westfall, Jon Freeman, Paul Reiche III
Joystick: Required
Format: Cassette

DATABASE

Here, courtesy of Commodore Business Machines, we publish another instalment of a detailed analysis of the Commodore 64's memory map.

LABEL	HEX ADDRESS	DECIMAL LOCATION	DESCRIPTION
FACHO	0062–0065	98–101	Floating Accum. #1: Mantissa
FACSGN	0066	102	Floating Accum. #1: Sign
SGNFLG	0067	103	Pointer: Series Evaluation Constant
BITS	0068	104	Floating Accum. #1: Overflow Digit
ARGEXP	0069	105	Floating-Point Accumulator #2: Exponent
ARGHO	006A–006D	106–109	Floating Accum. #2: Mantissa
ARGSGN	006E	110	Floating Accum. #2: Sign
ARISGN	006F	111	Sign Comparison Result: Accum. #1 vs #2
FACOV	0070	112	Floating Accum. #1. Low-Order (Rounding)
FBUFPT	0071–0072	113–114	Pointer: Cassette Buffer
CHRGET	0073–008A	115–138	Subroutine: Get Next Byte of BASIC Text
CHRGOT	0079	121	Entry to Get Same Byte of Text Again
TXTPTR	007A–007B	122–123	Pointer: Current Byte of BASIC Text
RNDX	008B–008F	139–143	Floating RND Function Seed Value
STATUS	0090	144	Kernal I/O Status Word: ST
STKEY	0091	145	Flag: STOP key / RVS key
SVXT	0092	146	Timing Constant for Tape
VERCK	0093	147	Flag: 0 = Load, 1 = Verify
C3PO	0094	148	Flag: Serial Bus—Output Char. Buffered
BSOUR	0095	149	Buffered Character for Serial Bus
SYNO	0096	150	Cassette Sync No.
	0097	151	Temp Data Area
LDTND	0098	152	No. of Open Files / Index to File Table

© 1983 ENTIS, G.—PACIFIC DATA IMAGES